Fall 12-23-2021

Evaluation of GPU Acceleration for WRF–SFIRE

Joshua Benz
San Jose State University

Evaluation of GPU Acceleration for WRF–SFIRE

CS 298 Project Report

Presented to

Prof. Ben Reed

Prof. Adam Kochanski

Prof. Robert Chun

Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Joshua Benz

December, 2021

Abstract

WRF–SFIRE is an open source, atmospheric–wildfire model that couples the WRF model with the level set fire spread model to simulate wildfires in real time. This model has many applications and more scientific questions can be asked and answered if the model can be run faster. Nvidia has put a lot of effort into easing the barrier of entry for accelerating applications with their tools to be run on GPUs. Various physical simulations have been successfully ported to utilize GPUs and have benefited from the speed increase. In this research, we take a look at WRF-SFIRE and try to use the Nvida tools to accelerate portions of code. We were successful in offloading work to the GPU. However, the WRF-SFIRE codebase contains too many data dependencies, deeply nested function calls and I/O to effectively utilize the GPU's resources. We look at specific examples and try to run them on a Titan V GPU. In the end, the compute intensive portions of WRF-SFIRE need to be rewritten to avoid data dependencies in order to leverage GPUs to improve the execution time.

ACKNOWLEDGMENTS

I would like to thank my project advisor, Dr. Benjamin Reed for setting me up with this project and all of the resources that I would need to complete this research. He has always been an inspiration for me ever since I took his distributed systems class. I knew right away that I want him to be my advisor. I couldn't have done it without his support. I appreciate all of his time, patience and effort in taking on this endeavor with me. I would like to thank my committee members, Dr. Robert Chun and Dr. Adam Kochanski for their support and for taking the time to do this with me.

A special thank you for the folks over at openwfm.org for maintaining this open source project that really benefits all of humanity in more ways than one. They made themselves available to me when ever I needed help. I am especially thankful for Dr. Adam Kochanski and Dr. Angel Farguell for all their support and providing me with the many needed resources to make this research possible as well as provide themselves as resources.

I would also like to thank all of the faculty and staff at SJSU. I have enjoyed everyone of my courses over the last few years at SJSU. Thank you to the Systems group and my peers at SJSU, in particular, Jonathan Wang. Of course, thank you to my family and extended family for supporting me while I was pursing this degree in San José.

Contents

List of Figures

List of Tables

I. INTRODUCTION

WRF–SFIRE is an open source, atmospheric–wildfire model developed by the Open Wild Fire Modeling Community (OpenWFM.org) based on the work of Jan Mandel, et al [16] [11][17]. It is made up of a mesoscale numerical model known as the Weather Research and Forecasting (WRF) model [18] coupled with the level set method fire–spread model to provide accurate, faster than real time wildfire forecasting [19]. WRF is described as a "next-generation mesoscale numerical weather prediction system designed for both atmospheric research and operational forecasting applications" [18]. The coupled WRF-SFIRE model showed promising results when compared to experimental burns, according to Adam K. Kochanski et al. [20], but the long term goal is to develop the model for "operational real-time fire prediction". The utility of WRF-SFIRE goes beyond that of fire spread simulation. It is also used for smoke forecasting [21][22] [23], air quality simulations [24] [25] and assessing socioeconomic impacts of fires [26].

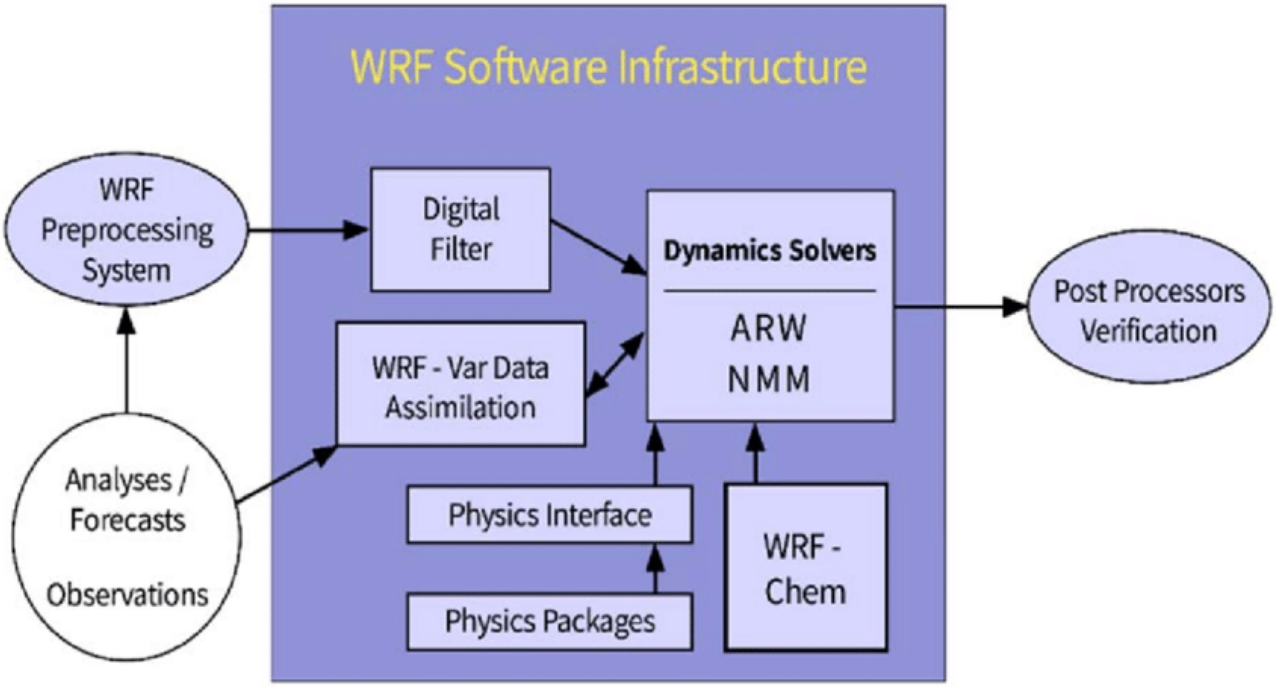

Fig. 1. WRF Software.
Source [1]

The software architecture of WRF is designed to be modular, flexible and scale through parallelization [1]. It decomposes the work to be done into segments so that it can leverage libraries like OpenMP and the Message Passing Interface (MPI) to provide shared memory parallelism, distributed memory parallelism or a hybrid of both. By using WRF as the base, WRF-SFIRE automatically has access to these features. According to benchmarks from the Computational & Information Systems Lab (CISL), WRF has

a linear strong scaling and thus benefits from scaling out to large core counts. However, as it scales out, the overhead associated with initialization, decomposition and input/output (I/O) also increases which can become a limiting factor [27]. They also suggest that the hybrid model, such as OpenMP with MPI, is best for utilizing every bit of performance possible, but it is still only marginally better than using MPI by itself. In recent years, researchers have been exploring a different hybrid model that utilizes Graphics Processing Units (GPU) to further exploit WRF's parallelism [28] [29] [30] [31][32] [33].

Over the years, Nvidia has developed a parallel computing platform and application programming interface called CUDA for leveraging GPUs for general purpose computing. Prior to the development of CUDA, utilizing GPUs for applications other than rendering to a screen was difficult and inflexible [34]. However, CUDA programs have become easier to write and have forward compatibility. More and more features are being added with the aim of easing the burden of writing code that can be run on GPUs.

II. RESEARCH OBJECTIVE

The objective of this research is to a try to improve WRF-SFIRE performance by utilizing GPUs and to evaluate the benefits or drawbacks of this hybrid model. WRF-SFIRE scales on large core counts but inherits the same bottlenecks as WRF. GPU's have the benefit of having thousands of cores on a single die and are designed to utilize parallelism through the stream programming model. According to Nvidia the keys to High Performance Computing (HPC) are efficiency of computation and efficiency of communication [34]. WRF-SFIRE handles the efficiency of computation by scaling out with MPI and OpenMP, but the overhead associated with decomposition, I/O and communication across a cluster of machines could be improved by utilizing the large core counts found in GPUs since they are on a single chip. GPU general-purpose computing (GPGPU) has become popular over the years since Nvidia's CUDA architecture because of its simple hybrid approach to mapping algorithms from the CPU to the GPU [5]. The goal is to apply this hybrid approach using current technologies, such as the joint Portland Group (PGI) and Nvidia HPC source development kit (SDK) [35], and determine if pieces of the work can be reasonably offloaded to the GPU without significant code or design modifications to the project.

III. LITERATURE REVIEW

A. Parallelism

Parallelism is a property of the application being written. The way that it is written can influence the amount of parallelism, but ultimately it is the application itself and its domain that determines how much parallelism is available.

There are various forms of parallelism including Bit-level, Instruction-level, data, and task parallelism. Bit-level parallelism involves manipulating multiple bits in a single clock cycle. Instruction-level includes utilizing multi-stage pipelines so that multiple instructions can be partially overlapped allowing for them to be in a different stage of the process at the same time. For example, while an instruction is being executed, another instruction can be fetched from memory. From a software perspective, we usually deal with data and task parallelism. Data parallelism is when data is distributed across computation nodes for the work to be done in parallel. Finally, task parallelism distributes tasks across computation nodes. Note that data parallelism executes the same task on multiple subsets of data in a synchronous, lockstep manner while task parallelism allows asynchronous execution of different tasks on the same or different data.

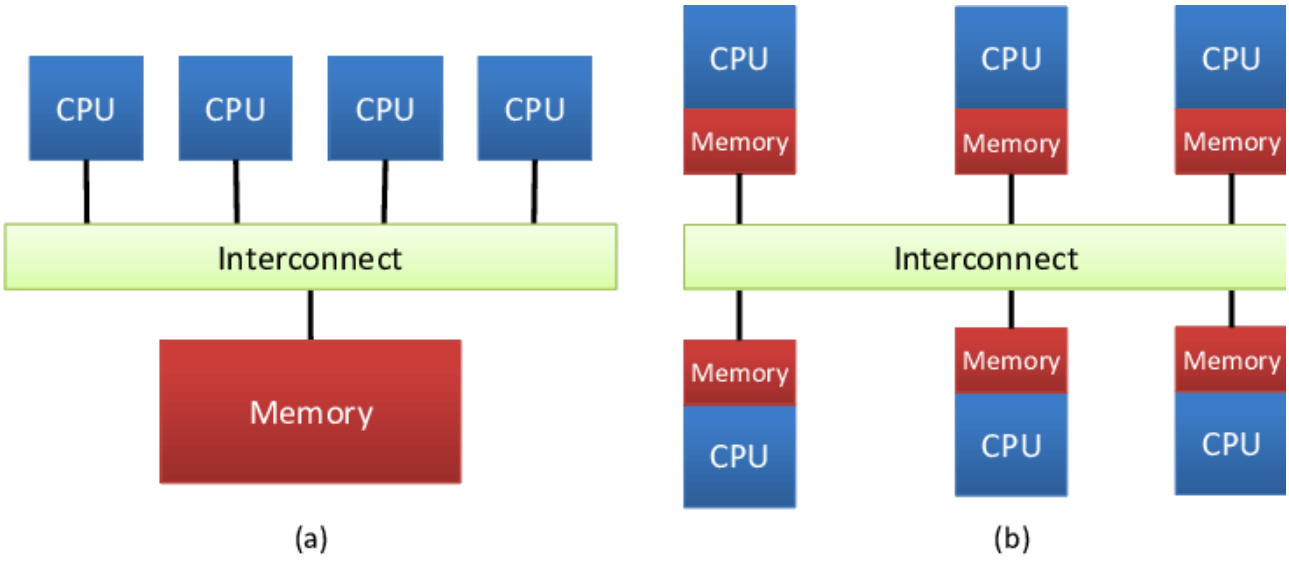

Fig. 2. Shared Memory (a) vs. Distributed Memory (b).
Source: [2]

One of the determining factors of what type of parallelism will be available is the memory model being used. There are three dominant memory models, distributed memory, shared memory and a hybrid of the two. Shared memory utilizes a global memory that can be accessed by all processors while distributed memory contains memory that is local to each processor and cannot be accessed by other processors. With shared memory, communication is implicit since all processors are accessing the same values in the same memory. However, theses accesses need explicit synchronization to avoid race conditions and other undefined behavior. Distributed memory, on the other hand, often uses message passing for explicit communication of data. The amount and type of parallelism available to a distributed memory model is dependent on the decomposition of data and the assignment of the work to be done.

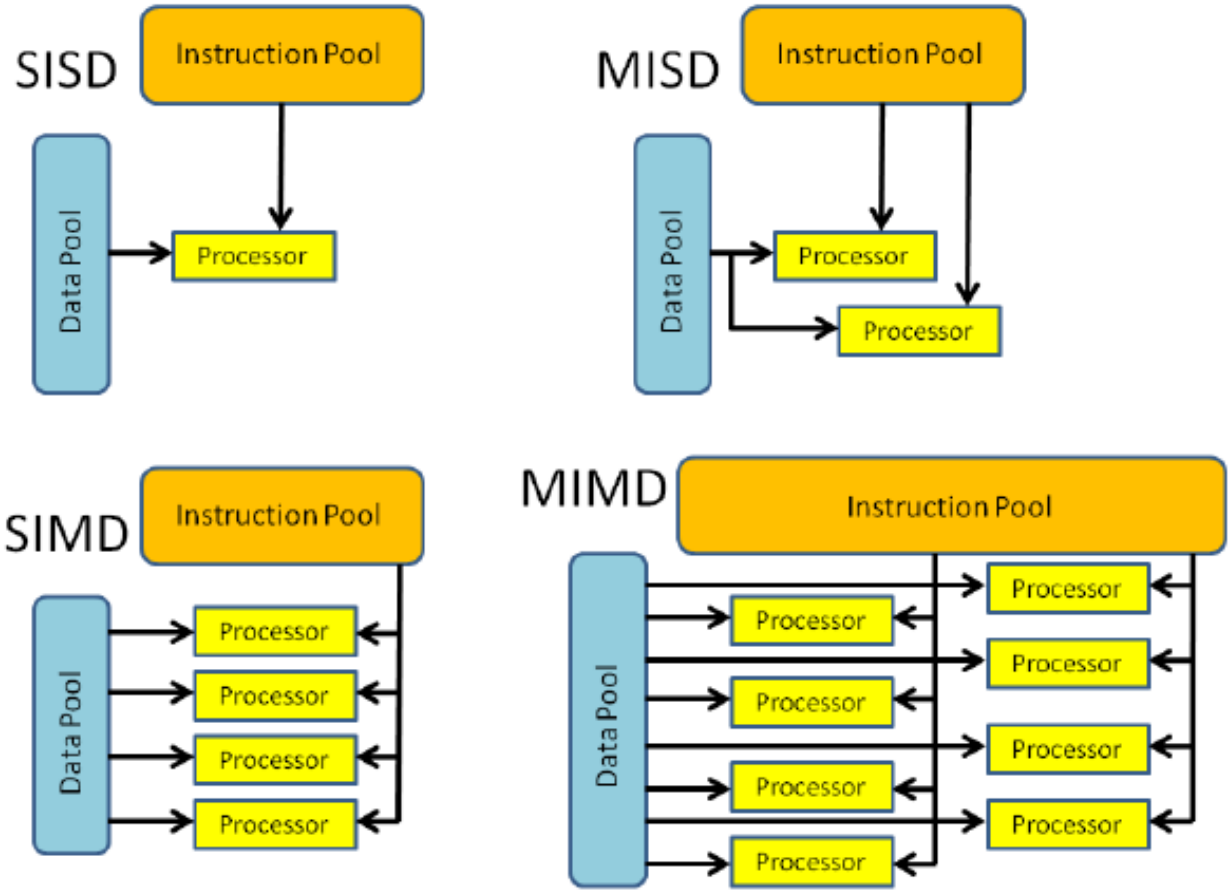

Fig. 3. Flynn's Taxonomy.
Source: [3]

Another one of the defining factors for the parallelism of a system is the architecture used. Michael Flynn [36] analyzed parallelism from the viewpoint of an assembly programmer. In his work he defines parallel architectures that perform concurrent operations on streams of instructions and data. He defined four classes of parallel architectures, Multi-Instruction Multi-Data (MIMD), Single-Instruction Multi-Data (SIMD), Multi-Instruction Single-Data (MISD) and Single-Instruction Single-Data (SISD).

MIMD is an architecture that achieves parallelism by connecting multiple processors together to perform operations independently of each other in a shared-memory or distributed-memory environment. Multiple programs can run simultaneously by executing different instructions and operating on different data. SIMD leverages data parallelism by coordinating all processors to execute the same instruction on different subsets of data in a synchronous lockstep manner. This can be done in either a shared or distributed memory environment but is less flexible due to its synchronous lockstep execution. MISD requires specialized hardware that allows multiple processors to execute different instructions on the same data. Finally, MIMD

systems allow processors to independently execute multiple instructions on different data [36]. The type of parallelism that GPU's exploit is most closely related to SIMD. In this research, the GPU in use is a Volta micro-architechture GPU. [37].

B. OpenMP

OpenMP is a shared-memory, data parallel application programming interface (API) that provides constructs to allow programmers to incrementally define the parallel execution of regions of code on multiprocessors that share memory and data. The goal is to simplify the specification of parallelism in an application. According to the documentation [38], OpenMP allows loop-level parallelism, nested parallelism and task parallelism. However, using OpenMP to accelerate applications comes with costs that are associated with doing things in parallel in a shared-memory context. The shared-memory model contains overhead from threads needing to synchronize, possible wasted idle time from imbalanced workloads, and runtime thread management such as the creation and destruction of threads [39].

OpenMP has been used to increase efficiency in many different types of applications that expose data parallelism. Increasing the efficiency allows for more questions to be asked and more answers to be revealed, particularly when applied to physical simulations. For example, J. Neal, T. Fewtrell and M. Trigg [40] use OpenMP to implement a parallel version of the LISFLOOD-FP hydraulic model. This is a flood model developed by P. D. Bates and A. P. J. De Roo [41] and is used in various industries to asses flood risks. According to J. Neal, T. Fewtrell and M. Trigg, the model was used over the years to simulate larger and larger areas up to the continental scale. In the cases of simulating large areas or simulating small areas in great detail, the simulation time grew until it was not feasible to simulate. The authors noted that the parallel implementation was simple to implement because of the ease of using OpenMP and it had a 5.8 times speedup on eight cores. This speedup allows the for the model to be pushed even further, answering more scientific questions along the way. This a recurring theme for physical simulations and algorithms in general.

A. Afzal, Z. Ansari and M. K. Ramis [42] profiled a conjugate heat transfer and fluid flow model that uses a computationally expensive technique that took up to 90 percent of the execution time when solving the pressure correction equation. They developed a parallelized version of the finite volume method using OpenMP. The authors were able to speedup the model by 1.5 times by using OpenMP. The authors profiled the application and found where their application was spending the most time. They then applied OpenMP to that region and was able to achieve some speedup. Their technique for finding which code to parallelize was used in this paper's technical approach to find regions of code that were worth parallelizing.

OpenMP has been used to accelerate general algorithms as well. M. Aznaveh et al. [43] apply OpenMP to the SuiteSparse implementation of GraphBLAS. GraphBLAS is a framework for creating graph algorithms based on mathematics or sparse matrix operations on a semiring. The authors use the GraphBLAS API to create various algorithms such as Bellman-Ford, k-truss, and clustering algorithms. Utilizing the API makes it easier to create these algorithms and you get the benefits of GraphBLAS without needing to write too much code. The authors used OpenMP to accelerate GraphBLAS which in turn accelerates the algorithms that get created using it. They ran and benchmarked six algorithms using real world data in their matrices on a 20 core machine. The amount of speedup depends on the structure of the graph which depends on the data being used. However, when compared to their single thread counterpart, algorithms such as Triangle Counting, 4-Truss, Breadth First Search, Bellman-Ford and Local Clustering Coefficient achieved speedups of up to 26.6, 27.7, 9.7, 18.9 and 30.2 times respectively.

C. MPI

The Message Passing Interface (MPI) is a specification for developers to utilize message passing in a shared or distributed memory environment. In distributed memory systems, data is moved between the local memory of cooperating processes. MPI is particularly useful when the data movement needs to occur over a network. Like OpenMP, MPI tries to expose a useful API while take care of all of the complexities of message passing [44]. According to the documentation, MPI is most suitable for MIMD architectures. MPI achieves parallelism by having each process execute the same or different program, independent of the other processes or programs, and communicate the necessary data to each other. B. Armstrong, S. W. Kim and R. Eigenmann [45] evaluate the performance impacts from overhead associated with OpenMP and MPI. They found that MPI in a shared memory environment is performance equivalent to OpenMP. MPI is typically used for applications that are distributed across a cluster that is connected by a network for communication, such as a high performance computing cluster (HPC).

V. Tan and L. Hluchy [46] discussed parallelizing sequential, numerical flood models using MPI. They pointed out a few benefits of this approach that incorporated this approach into my research as well. First, despite the mathematical approaches being well-known, there are difficult, algorithmic details contained in the models that they used, namely FESWMS and DaveF. By using MPI, there was no need to understand the intricate details of these algorithms. The programmers only needed high level knowledge of how the models worked. Second, since they did not need to understand the algorithms in detail, there was no need to understand all $65,000$ lines of their implementation. Instead, they utilized profiling tools to identify compute-intensive regions of the source code and spent that time becoming familiar with the important

data structures. Finally, they needed to modify the routines that they identified using the profiling tools to leverage MPI. They ended up modifying about 50 lines of code in the FESWMS model and 100 lines in the DaveF model and ran experiments on a Linux cluster. They found that they have a $5-7$ times speedup. The approach used in this authors' research was applied to the technical approach used in this paper. In other words, the regions of code to parallelize in WRF-SFIRE were found by utilizing the profiling tools, obtaining general knowledge of the algorithms and data structures and applying parallelism to the compute-intensive routines.

D. MPI and OpenMP Hybrid

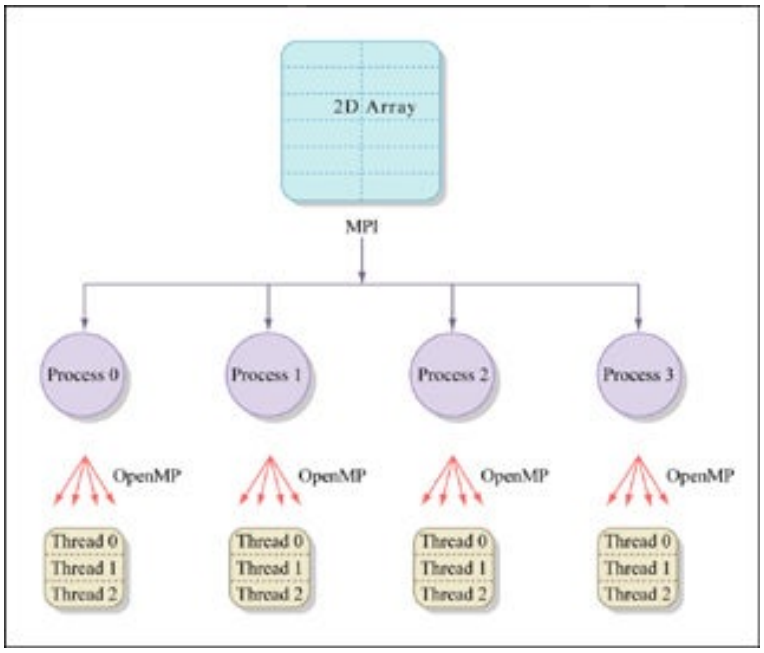

Fig. 4. MPI-OpenMP Hybrid.
Source: [4]

It is fairly common to see MPI used in conjunction with OpenMP, particularly on HPC clusters. MPI was used to distribute the work across the nodes in the cluster and OpenMP was used to utilize the resources of each local machine. F. Cappello and D. Etiemble [47] researched which programming paradigm would be superior to the other by running benchmarks on two IBM SP systems, which is a clustered super computer with 604 nodes [48]. They noted that the superiority of one model over the other depended on the level of shared memory parallelization, the communication patterns and the memory access patterns. They were comparing a unified MPI model with a hybrid MPI and OpenMP model. They discovered that the speed of the hardware components of the machines in the cluster made a difference when selecting an approach. They found that the Unified MPI model performed better than the hybrid model except for when

the speed of the processors in the cluster was faster than the overhead of communication and when there was a sufficient amount of parallelization [47]. This seemed to suggest that using the hybrid model was worth the overhead when the resources of each local machine can be fully utilized and if those resources were faster than the network link.

Research conducted by B. Yan and R. A. Regueiro [49] confirmed this as well. They were computing 3D Discrete Element Method (DEM) simulations. They found that they had great success with OpenMP when they applied it to specific parts of their code, but the unparallelizable regions of code forced them to have lower granularity which led to less MPI network calls which made the MPI approach outperform the hybrid approach.

E. GPGPU

GPUs and GPGPU have been making a lot of progress over the years to ease the barrier of writing code that can be ran on the GPU to improve performance. It has become popular in research and applied to many different areas such as mathematics, simulations, digital signal processing, database operations, molecular dynamics, artificial intelligence and high performance computing.

G. Chen, G. Li, S. Pei and B. Wu [50] took advantage of the lower barrier that GPU developers had created and wrote Molecular Dynamics (MD) simulations that could run on the GPU. The authors used an AMD GPU and their Brook+ streaming environment to implement a parallel version of their MD algorithms. They managed to achieve speedups of up to 15 times that of the sequential version.

A. J. Kalyanapu, S. Shankar, E. R. Pardyjak, D.R. Judi and S. J. Burian [51] implemented a parallelized version of a 2D flood model. They leveraged Nvidia GPUs and CUDA in an attempt to speedup the full dynamic wave flood model and validate the results from observations. They found that their GPU accelerated Taum Sauk flood simulation was 80 to 88 times faster than the sequential implementation. This allowed them to cut the simulation from thirty minutes to two minutes and fifteen seconds. They found that the GPU implementation scaled well and that it could be applied to other flood modeling techniques.

S. Aydin, R. Samet and O. F. Bay [34] applied GPGPU to image processing. In their work, they aimed to speedup image processing by segmenting the image using a thresholding technique in parallel. One of the interesting parts of this research was that they explored different methods of transmitting data to GPU memory. First they tried to transmit the images one by one, then distribute each pixel to a core in the GPU and return the resulting images one by one. The second technique was to transmit the images one by one to the GPU, distribute groups of pixels to each core and return the resulting images one by one. The third technique was to combine the images into a data array, distribute a single pixel to each core and

return the results as a combined data array to be separated by the CPU cores. The final and most efficient technique was to combine the images into a data array, distribute groups of pixels to each core and return the results as a combined data array to be separated by the CPU cores. They found that the final technique was the fastest and performed 16 times better than the serial algorithms. This research seemed to suggest that sending large amounts of data all at once outperformed sending data one at a time. They also show that it was important to give enough work to each GPU core to maximize the efficiency.

F. GPU Acceleration of Other Atmospheric Models

One of the necessary properties for a parallel algorithm to be more efficient than its serial counterpart is the amount of parallelism that can be made available in the problem. Many types of applications that simulate physical processes have been successfully made faster by utilizing GPGPU. One of the major functions of WRF-SFIRE is to simulate atmospheric conditions, therefore, it is beneficial to look at the literature to see if this type of physical process can be efficiently accelerated with GPGPU.

R. Kelly [52] had the same question in 2010. The author noted that GPGPU works well for applications that are data parallel or have small, compute intensive regions of code that can run on the GPU. Kelly explored the possibility of applying GPGPU to large scientific models by trying to accelerate the Community Atmospheric Model (CAM) from the National Center for Atmospheric Research (NCAR). Fortran wasn‘t supported by Nvidia yet, so the author ported a section of the code to the C language in order to utilize CUDA. The main data structures consisted of a grid of 3D cells that contained latitude and longitude information from the surface up into the atmosphere. The fortran code represented this information as multiple 2D arrays while the author represented it as 1D arrays. The author managed to obtain a 14 to 20 times speedup when compared to single core sequential performance of that particular region of code. At the end of the research, Kelly discussed what it would take to accelerate the entire CAM model. It was noted that most of the CAM model would need to be run in the GPU in order to make the cost/benefit worth the effort. Kelly also concluded that it would be ideal to use a cluster of GPUs.

P. E. Bieringer et al. [53] describe the advancements made in Large Eddy Simulation (LES) for atmospheric transport and dispersion of pollutants at spatial and temporal resolutions that were needed for various applications such as assessing the impact of airborne pollutants on human health. These models remained at the proof-of-concept stages because the computational requirements for running these simulations were unfeasible. These models stayed in this stage until they could be accelerated with GPUs. They describe one such GPU-based model called JOULES and found that it is 150 times faster than CPU based simulations.

I. Demeshko, N. Maruyama, H. Tomita and S. Matsuoka [54] presented a partially GPU accelerated version of the Nonhydrostatic ICosahedral Atmospheric Model (NICAM) and compared the performance of the MPI parallel version to the GPU accelerated version on a multi-GPU system that consisted of 320 GPUs. The MPI version uses 2D decomposition to divide the grid into regions and distributes those regions to MPI processes. Their approach was to copy the constant data arrays all at once to the GPU, and copy the "One large step" data arrays at the beginning of each step. The number of GPU accelerated kernels was equal to the the number of MPI processes. This work also took a look at communication between the CPU and GPU. They found that that the CPU-GPU communication was their bottleneck and eased that communication bottleneck by utilizing pinned memory, which allowed them to achieve near maximum bandwidth, reducing the communication time by 3 times when compared to the non-optimized version. Despite the communication optimizations, they found that the code spent $30-40\%$ of its time in CPU-GPU communication. The authors tested their implementations with multiple cluster configurations. Their largest comparison was 107 nodes with 1280 CPU cores for the original MPI implementation versus 320 CPU-GPU hybrid cores. They found that the CPU-GPU hybrid implementation is three times faster than the highest performing MPI version. They analyzed and estimated that a full GPU implementation would be 4.5 times faster than the MPI version.

J. Zeng, T. Matsunaga and H. Mukai [55] presented a GPU accelerated version of a Lagrangian particle dispersion model based on FLEXPART. This model computed trajectories of particles to describe the transport and diffusion of tracers in the atmosphere. They found that computing the position changes of particles was the bottleneck in the CPU based implementation. In this particular case, the researchers decided that they wanted to write the GPU code such that it could run on the CPU with minimal modifications. This came at the cost of using global GPU memory for everything, which tends to be slower than using memory that is more localized to the cores. Nonetheless, they were able to speedup the application up to 10 times that of its sequential counterpart.

M. C. dos Santos, C. M. N. A. Pereira, R. Schirru, and A. Pinheiro [56] describe the use of amospheric radionuclide dispersion systems (ARDS) to predict the the outcomes of unexpected radioactive materials being released from nuclear power plants in order to design emergency preparedness plans for people living near them. The authors picked one of the main four modules that required heavy compute resources and implemented a GPU version of that module. In this case, they designed a GPU-based plume dispersion module. They compared the CPU and GPU functions that performed the calculations and found that the GPU version had a speedup of 11.63 times . A. Pinheiro, F. Desterro, M. Santos, C. Pereira and R. Schirru [57] created a GPU version of the Wind Field module, which was another one of the compute heavy modules in ARDS. They were able to achieve a 40 times speedup when compared to the sequential version.

Since these two implementations were successful, the authors of [56] were confident that the entire ARDS model can benefit from porting other portions of the model to be GPU accelerated as well.

M. Govett et al. [58] described the Non-Hydrostatic Icosahedral Model (NIM) and implemented a parallel version to demonstrate that weather prediction models can be designed to be highly parallel. They compared the performance of the model using CPUs, Many-Integrated Core (MIC) processors and GPUs. They utilized MPI and OpenMP to accelerate the CPU portions and OpenACC to accelerate GPU portions. They found that this model can be scaled to thousands and tens of thousands of compute nodes. The authors implemented micro-physics routines such that the same code could be executed on CPUs, MICs and GPUs. They found that the MICs and GPUs had a 2 and 2.5 times speedup respectively.

G. GPU Acceleration of WRF Modules

GPGPU has been the driving force behind Nvidia's CUDA hybrid model where the CPU and GPU are used simultaneously. This also makes it easy to incrementally port existing code to the GPU, which seems to be the approach in the literature when trying to GPU accelerate pieces of WRF.

M. Huang et al. [30] found that they could obtain a speedup of 311× that of a CPU core when leveraging a Tesla K40 GPU to process one of the WRF weather models called the five-layer diffusion scheme. This is a weather model that had good parallel properties since there were no interactions among horizontal grid points. The authors found that they gained significantly more speed up by adding a GPU than the 3.1 speedup from adding another CPU. They also found that adding a second GPU increased the speedup to 398 times when compared to single core performance.

In 2008, J. Michalakes and M. Vachharajani [59] showed that by accelerating a computationally intensive portion of WRF, they could achieve an overall speedup. They utilized CUDA to implement a GPU-based version of the WRF Single Moment 5-tracer (WSM5) microphysics module. Rather then rewrite the Fortran code in C they managed to use experimental Perl-based processor directives to create their kernel. One thing they took into account that will also occur in WRF-SFIRE is the size of the data structures. They could not fit all of the necessary data into local memory for a thread-per-column decomposition. They needed to use the slower shared memory in order to fit all of the data, which influenced their design decisions. As a result, they paid special attention to how they could allocate shared memory for reuse. In the end, they managed to speed up WSM5 by 8 times, which in turn allowed the entire WRF model to have a speedup of 1.23 times. C. El Amrani and I. M. Hedgecock [60] encountered similar problems to [59] when they ported the WRF-Chem model to utilize grid computing and GPUs. They found that WRF-Chem processing took a long time and required a lot of storage space. For both of these authors, storage space became a limiting

factor what and simply noted that better memory management would be a beneficial endeavor in the future.

In [28] Mielikainen et al. developed a GPU accelerated WRF kessler microphysics scheme and obtained a speedup of 70× over the serial CPU execution. In [32] Mielikainen et al. GPU accelerated the computation of WRF Double-Moment 6-class Microphysics scheme (WDM6) and achieved a 150× speedup over the single threaded execution. Excluding I/O transfers, the speedup was 206× and when four GPUs were used, the execution obtained a 715× speedup. Mielikainen et al. [31] managed a speedup of 1556× without I/O and 206× with I/O when accelerating the WRF Single Moment 5-Class Cloud Microphysics (WSM5) model on four GPUs. In [61] Mielikainen et al. accelerated the Goddard Shortwave Radiation Scheme and achieved speedups of 536× and 259× with and without I/O respectively when using two Nvidia GTX 590 GPUs. Others such as Silva et al. [62] and Huang et al. [63] have put considerable work into trying to GPU accelerating the entire WRF model with promising results.

Y. Wang, Y. Zhao, J. Jiang and H. Zhang [64] implement a GPU–based Longwave Radiative Transfer model (RRTMG_LW). When simulating the atmospheric physics model, the radiative process is used for calculating radiative fluxes and heating rates. HPC technology has helped the radiative transfer take less time to compute. The authors' goal was to further accelerate the model using CUDA Fortran. They proposed RRTMG_LW which utilizes a 2D domain decomposition that exposes more parallelism than the previous iterations of the algorithm. They tested the resulting model by comparing it to the results of simulating an ideal global climate for one simulation day. They managed to achieve a 30.98 times speedup when compared to single core performance.

M. Huang, J. Mielikainen, B. Huang, H. Chen and M. D. Goldberg [63] implement an accelerated Yonsei University (YSU) scheme for the WRF planetary boundary layer (PBL) model. The PBL is the lowest part of the atmosphere that is affected by its contact with the planetary surface. It is responsible for providing a model for atmospheric temperature, moisture and horizontal momentum. It does this by accounting for "vertical sub–grid–scale fluxes due to eddy transports in the whole atmospheric column". The authors managed to obtain a speedup of 193 times when compared to single core performance and 360 times by adding a second GPU.

IV. BACKGROUND

A. CPU vs GPU

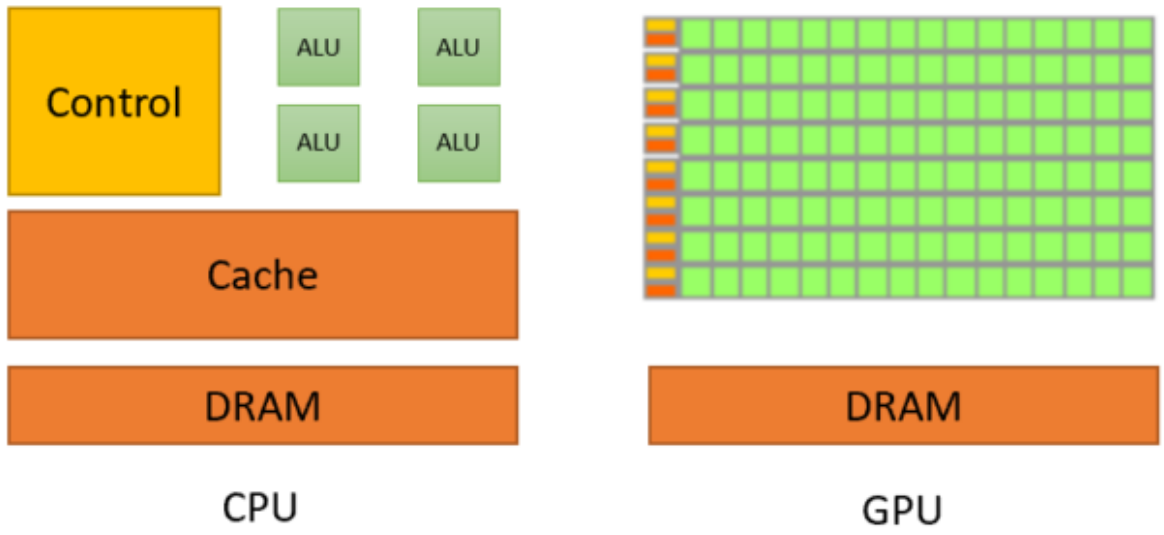

Fig. 5. CPU vs GPU Hardware Configuration.
Source: [5]

CPUs and GPUs handle parallelism in different ways because of their design and intended use case. GPUs were created for a graphics rendering pipeline while CPUs are generalized so that they can perform any task. The consequences of their use cases affect the architectural design behind their hardware. CPUs need to be able to handle complex control workflows because it is the central hub of controlling a system. Therefore, it has a lot of hardware dedicated to handling this control. Memory accesses are expensive, so data is kept as close as possible to the chip in the forms of cache and the memory systems are designed to minimize latency. The problems that CPUs are used to solve are limited in the amount of parallelism that they can perform. Many tasks are serial in nature and must be done serially. GPUs on the other hand solve specialized problems that are parallel in nature. They have specialized hardware for certain types of tasks, which outperform CPU hardware at these particular tasks.

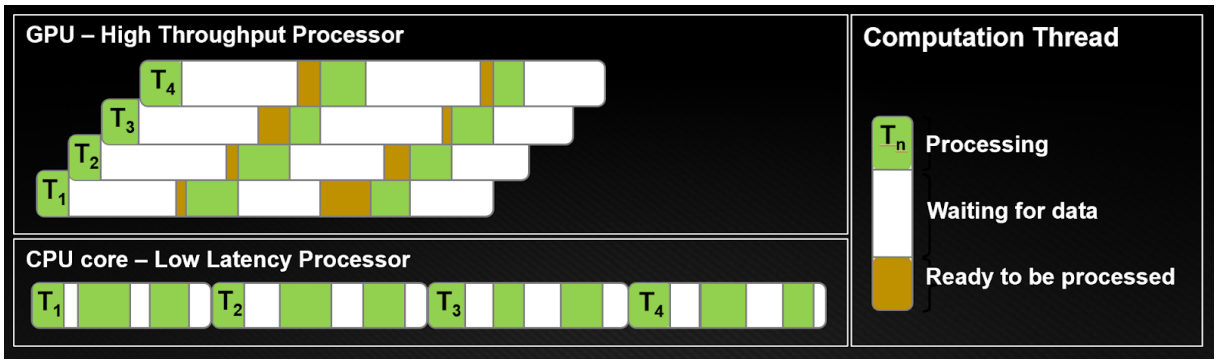

Fig. 6. GPUs try to maximize throughput.
Source: [5]

The GPU stream processing execution model and memory systems are designed to maximize throughput over latency.

B. GPU Hardware

It is helpful to visualize the physical hardware of a GPU before talking about the logical divisions made in CUDA when allocating resources. The GPU used for this research was a Titan V which is a Volta micro-architecture GPU. Most of the GPU families over the years have the same basic components in similar configurations. Figure 7 and Figure 8 are taken from one of Nvidia's blog posts [5] that introduces the Volta micro-architecture. Although their article is looking at a GV100 GPU, the underlying architecture applies to the Titan V as well. Understanding the hardware view will help make the computation allocations and logical divisions more clear.

Fig. 7. Volta architecture GPU .
Source: [6]

Figure 7, from Nvidia's whitepaper on the Volta micro-architecture, shows that the GPU is divided into Graphics Processing Clusters (GPC). Each GPC contains multiple Texture/Processor Clusters (TCP), which each contain two Streaming Multiprocessors (SM). Each one of these GPCs are connected to a high speed hub for communicating with each other as well as memory controllers to access global HBM2 memory. These components provide a flexible way of allocating resources for the pipeline to perform operations on streams of data.

Fig. 8. Volta architecture GPU SM view.
Source: [6]

Figure 8, from Nvidia's whitepaper on the Volta micro-architecture, zooms into a single SM. Each SM is divided into four computation blocks. Each block contains a local cache, dedicated registers in the register file, load/store units (LD/ST) and more. this architecture allows for simultaneous integer and floating point operations and contains a shared memory in the form of an L1 data cache [6].

C. Resources and Logical Divisions

The hardware organization of GPUs is what allows them to utilize the stream processing model and exploit multiple levels of parallelism. Thread processors, also known as cores, are the building blocks of the GPU. In Figure 8, the FP64, INT, FP32, etc are the individual cores. These cores are grouped together into computational blocks in an SM, which have their own registers and shared memory. When a GPU program is launched, each thread will execute that code on a subset of the data. These threads are grouped together into thread blocks. The thread blocks are further grouped into grids.

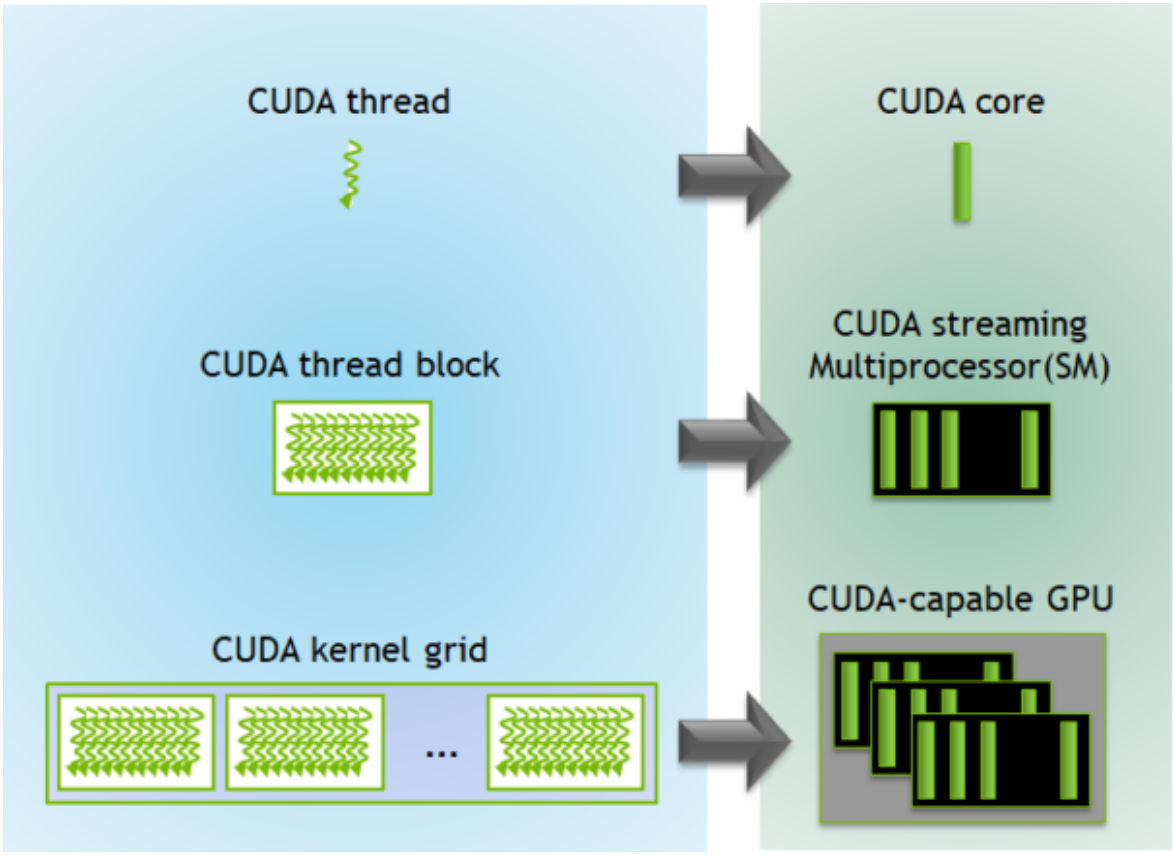

Fig. 9. Mapping to Hardware.
Source: [7]

As seen in Figure 9, Grids are distributed across GPUs while thread blocks in those grids are distributed to SMs. Finally, threads in the thread blocks are executed on the individual cores. However,

there is another logical division made by the GPU. Groups of 32 threads, called warps, are created and managed at a time. In other words, a warp is the unit of thread creation, management and scheduling. Warps will be discussed in more detail in subsection D.. GPUs have global memory, shared memory, local memory and constant memory. The device's global memory resides in the DRAM and is accessible by the host and to all threads on the device. Local memory also resides in the DRAM and is used to store local variables that cannot be stored in the registers on the device. Shared memory resides on multiprocessors and is available to threads in a thread block. Constant memory resides in DRAM and is made read-only to threads on the device. GPUs also have caches that usually cache data from global and local memory. [65].

D. CUDA Model

CUDA makes a distinction between the CPU and its systems and the GPU. The system that is launching the CUDA program, called a kernel, is called the host. Usually it is the CPU system that is launching the kernel, so this system is referred to as the host. The system that is executing the kernel, in our case the GPU, is called the device. There are three basic steps involved in running a kernel. First, there is a host-to-device data transfer that copies the inputs from the host memory to the device memory. Next, the kernel is launched and executed on the device. Finally, there are device-to-host data transfers that copy the results back the host memory.

A kernel is executed as a grid of blocks of threads. We can specify the computational resources allocated by using different dimensions for the grids and blocks. In a language like C, the dimensions of the grid are specified as $<<< blocks_{dim3}, threads_{dim3} >>>$, where *blocks* and *threads* can be up to three dimensions. This means that the resources can be allocated and indexed as a vector, matrix or volume. For example, $<<< 1, (N, N) >>>$ means that this grid contains one block with $N \times N + 1$ threads per block. This also means that a grid can execute multiple blocks of the same shape. These grid dimensions are important because they are what allows the CUDA runtime to provide the information needed for the programmer to access the correct subset of the data when executing a kernel.

Nvidia's CUDA architecture, according to Nvidia's book GPUgems2 [34], utilizes the stream programming model, which represents data as long streams. Kernels, are executed and chained together to form a pipeline to perform operations on these streams. There are two assumptions made when using kernels. First, the data required for kernel execution is known when the kernel is compiled. Second, stream elements must be computationally independent. This allows GPUs to expose data parallelism by doing work on hardware designed to be data-parallel. Memory is efficiently used because kernels minimize global memory accesses by copying the entire data stream into shared memory on the SM while also trying to avoid off chip

memory access. GPUs utilize task parallelism by mapping kernels to specialized functional units. Then the kernels can process data elements from the stream in parallel in the local execution context.

CUDA utilizes the single instruction, multiple threads (SIMT) model for parallel computation. This is similar to the single instruction, multiple data (SIMD) model found in Flynn's Taxonomy [36]. The unit of thread creation is a warp, which has a size of 32 threads. When an SM receives a thread block, it will partition the threads in the block into warps. SMs schedule execution of these groups of threads to execute the same instruction at the same time. Each thread in a warp gets its own program counter, registers and starts at the same starting address.

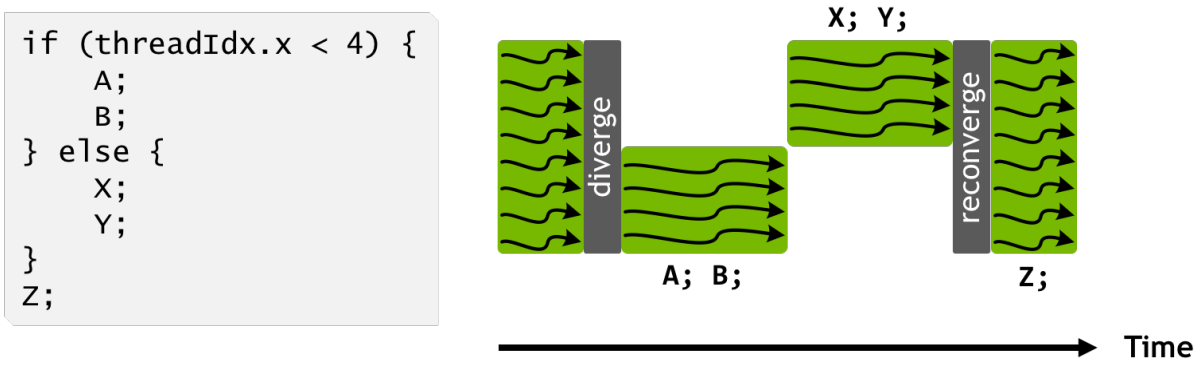

Fig. 10. SIMT execution evaluates one side of the branch serially, then the other side serially. Once both sides of the branch are completed, they execute all together again.
Source: [6]

The warp will execute in a lock-step manner until there is a divergence within the warp as seen in Figure 10. When there is a divergence from a branch or some other cause, the thread can execute independently at the cost of the other threads waiting for that thread to converge. However, the Volta architecture allows for thread independent scheduling which leads to interleaved scheduling. Although the current warp must wait for convergence of all the threads, the other warps in the SM can continue to run in parallel. The SIMT model exists because threads are mapped to hardware. Although these warps have more flexibility, it comes at the cost of having to wait if there is a divergence. For this reason, it is best to write code that doesn't branch often in order to maximize warp efficiency [37].

E. Generating Kernels with OpenACC

The authors of [58] made a good observation about the trends of GPU programming. In the scientific computing community, rewriting sections of code to perform on the GPU is bug prone and difficult. CUDA has made effective leaps in easing the barrier of entry for developing GPU accelerated programs. However, writing specialized GPU code is still a difficult task. OpenACC was developed to ease that barrier

of entry even further. It was specifically designed for accelerating regions of code to run on the GPU without needing to maintain a separate version of the source code and without significant source modifications.

The Nvidia–PGI HPC compilers support generating CUDA kernels from OpenMP and OpenACC directives [66]. WRF–SFIRE uses OpenMP to compute tiles in parallel. In order to avoid conflicts with existing OpenMP directives, OpenACC was used in this research to generate the CUDA kernels. OpenACC is a programming standard that provides directives for marking up code to simplify and leverage parallel computing. The goal is to use these directives to describe how a region and its data should be executed in parallel. The benefit of using something like OpenACC and OpenMP is that it is portable and works across CPUs, hosts and compilers.

According to the OpenACC specification [15], the host executes the kernels on a device such as a GPU. Once launched from the host, the GPU will execute parallel regions of code, which are typically loops that are marked as kernel regions. The host is also responsible for initiating data transfers, sending the kernel code to the GPU, passing arguments, queuing GPU code for execution, waiting for completion, deallocating GPU memory and even transferring data back to the host. OpenACC supports nested, fine-grained and coarse-grained parallelism. It is best to localize the fine-grained parallelism to rapidly switching execution among threads inside of an execution unit, such as an SM, to better tolerate long latency memory operations. They also offer support for SIMD operations within an execution unit.

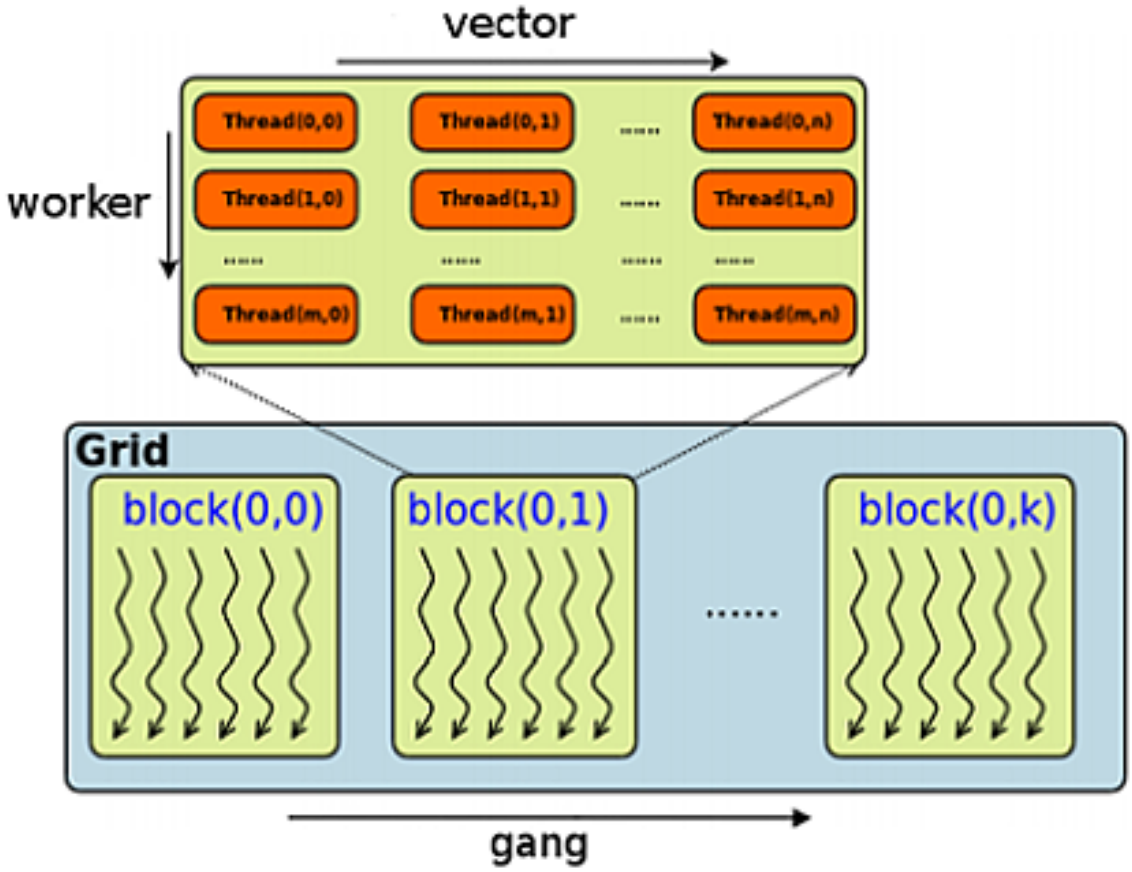

Fig. 11. Gangs, Workers and Threads.
Source: [8]

OpenACC exposes three levels of parallelism, gangs, workers and vectors. Gangs, which mapped to thread blocks in a GPU, work concurrently to provide a coarse-grained type of parallelism. Multiple gangs can be launched by the GPU. These gangs have one or more workers with shared resources. Workers map to CUDA warps. Workers have one or more vectors, which are equivalent to CUDA threads. Recall that CUDA threads are launched and managed as warps. Each warp has 32 threads which means that it would be ideal to have vector lengths be multiples of 32. Worker parallelism is fine-grained while vector parallelism is for vector operations within a worker, such as SIMD operations.

When the host launches a kernel, gangs are launched in the GPU. There are two modes that gangs can be ran in, gang-redundant mode and gang partitioned mode. All gangs begin in the gang-redundant mode which means that vectors in each worker executes the same code. When the kernel contains loops with work-sharing at the gang level, the kernel transitions the gangs from gang-redundant mode to gang-partitioned mode. In this case, loop iterations are distributed across gangs to be executed in parallel. Similarly, workers and vectors have modes for enabling work-sharing. When worker-single and vector-single

modes are enabled, then only a single worker and a single vector lane are doing the work. This means that if work-sharing is encountered at the gang level, the kernel transitions the gangs into gang-partition mode, but if there is no further work-sharing at the worker or vector level, then these levels will be executed in their respective single modes. However, if worker level work-sharing is encountered or vector level work-sharing is encountered they will enter worker-partitioned and vector-partitioned mode respectively. This means that the work is distributed among all of the workers and vector lanes [15]. These things are important to keep in mind when utilizing OpenACC so that the resources of the GPU can be effectively utilized.

Construct	Description
parallel	One or more gangs of workers are used to execute the region. Default is gang-redundant mode.
serial	Used to execute code sequentially using one gang with one worker and one vector lane
kernels	The region of code should be compiled into a sequence of kernels for the GPU. The compiler will try to determine what the best configuration of resources is for the regions.
private	This is used to tell the compiler to make a copy of a variable for each gang to own when used with parallel or serial.
firstprivate	This does the same thing as private except it guarantees that the copy will be initialized with the value in the local thread.
reduction	In a reduction a private copy of a variable is given to each gang. At the end of each of the gangs execution, the values from each gang will be combined with the original variable.

Table 1: Common Compute Constructs. More information can be found in the specification [15]

Note that for compute constructs, the kernel cannot branch into another parallel region. There is also no guarantee on the order in which the compiler will evaluate the constructs, so the developer must not rely on the order that they supply.

Construct	Description
create	Allocates memory in the device memory if the it does not exist in shared memory. Once the data goes out of scope, a present decrement or detach occurs.
copyin	Copies data into device memory if it does not already exist
copyout	Copies data out of device memory.
copy	Performs a copyin at the beginning of the data region and performs a copyout at the end of the data region.
present	This tells the compiler that the specified data is already in shared memory or device memory for the device to access.
delete	Deallocates data

Table 2: Common Data Clauses. More information can be found in the specification [15]

The OpenACC specification defines three attributes that are attached to variables, predetermined, implicitly determined and explicitly determined. OpenACC provides data clause directives for the user to

define where and how data will be moved in that region of code. Predetermined variables cannot appear in a data clause while explicitly determined variables must occur in a data clause. Similarly, implicitly determined may appear in a data clause if it overrides the implicit attribute. Each piece of data has an associated lifetime. Data that is in shared memory is available to the device as long as the data is in scope. Typically, the lifetime of data in shared memory is when the data is allocated and goes out of scope when deallocated. However, static lifetimes occur when the kernel starts and the data goes out of scope when it ends [15].

Construct	Description
data	Defines a data region with the lifetime of the region. This is used to specify a list of variables to be copied or created.
end data	This is used to mark the end of a data region
enter data	This is used to create or copy variables from the specified list into device memory for the remainder of the program or until an exit data is encountered.
exit data	This marks the end of a data region and can be used to define what data should be copied back to the host.

Table 3: Common Data Directives. More information can be found in the specification [15]

OpenACC recognizes that devices, such as GPUs, typically have their own discrete memory. The host cannot access device memory and the device cannot access host memory. OpenACC will try to implicitly perform these copies, but there are a couple of potential pitfalls. First, the memory bandwidth between the host and the device is the deciding factor for the level of compute intensity that is needed to accelerate code. Secondly, the discrete memory on a device is typically smaller than the host memory, so regions with large amounts of data would need to be broken up or are simply infeasible to perform on the device.

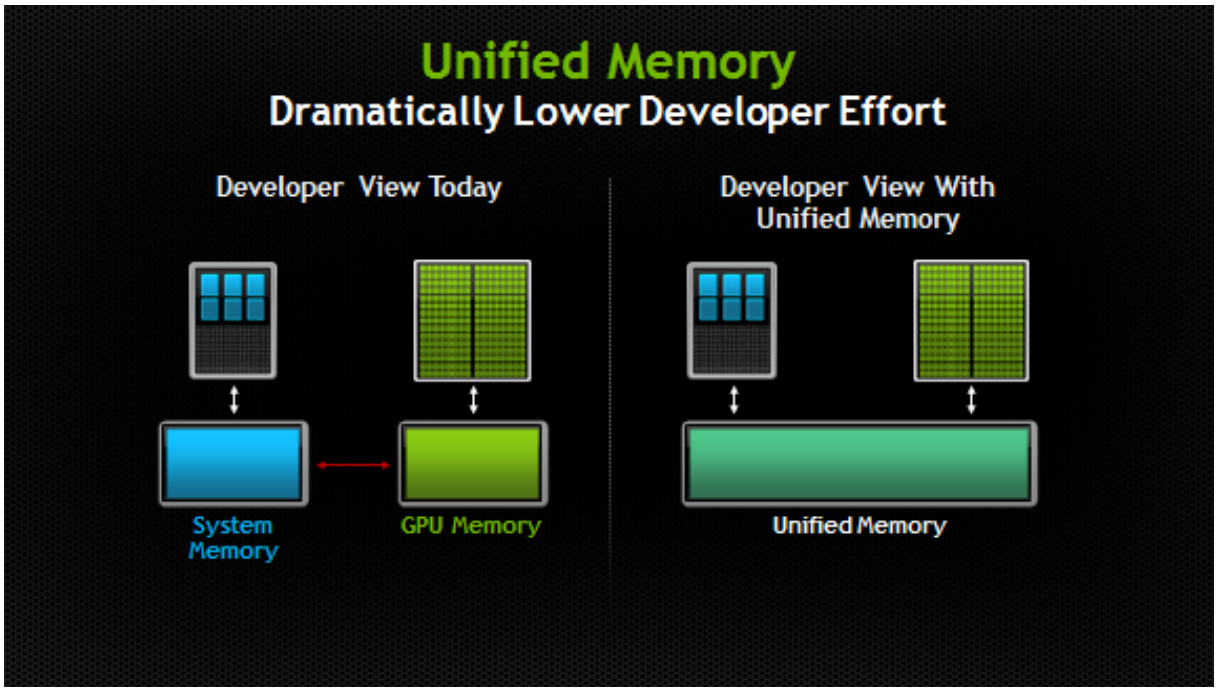

Fig. 12. Unmanaged Memory vs Unified Memory.
Source: [9]]

Finally, pointer values from the host are pointing to memory regions that exist only on the host. Accessing a pointer that points to host memory is invalid. Therefore, data that a host pointer points to should be explicitly copied into the device memory. This is most common when dealing with data structures that contain pointers. OpenACC must first copy the the structure itself. This will copy the structure and if there are pointers, it will copy those pointers even though they point to memory on the host. Once the structure is copied, the actual values that those pointers point to need to be explicitly copied into the device memory.

In recent CUDA architectures, this problem was eased by introducing unified memory. CUDA unified memory makes the host and GPU memory into a single address space for any processors connected to the system to see. This managed memory mode will enable the CUDA software or hardware to take care of transferring pages of memory to wherever they need to be [67][9]. Unified memory changes the task of memory management to be an optimization task. In this model, data is still moved but CUDA moves it implicitly.

1. CUDA-aware MPI

CUDA has this notion of CUDA-aware MPI which allows it to manage buffers differently depending on which memory the the data is residing. Similar to unified memory, CUDA can create a Unified Virtual

Address (UVA) space. This is a combination of memory between all of the hosts and GPUs in the cluster and using them in the same address space. This setup allows pipelining message transfers and utilizing Nvidia's GPUDirect to send data from one device memory directly into another device's memory without needing to stage in the host via Remote Direct Memory Access (RDMA)[68]. The configuration used for the WRF-SFIRE experiments utilized CUDA-aware MPI from OpenMPI, which comes with the NVIDA HPC SDK.

F. WRF-SFIRE

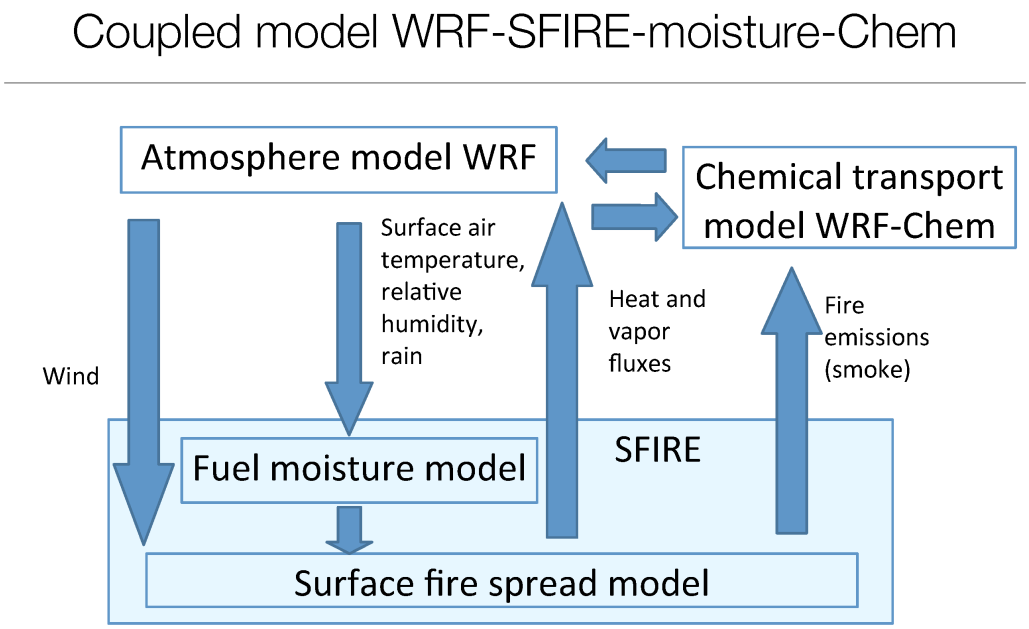

Fig. 13. WRF-SFIRE.
Source: [10]

J. Mandel, J. D. Beezley and A. K. Kochanski describe the WRF-SFIRE model in [11]. The authors note that wildfire behavior can be captured though a mesoscale weather model such as WRF and a simple fire spread model such as the level set method. One of the main drivers behind the spread of a fire is the wind. The fire itself influences the atmosphere via moisture and heat fluxes which in turn impact local wind circulations near the fire front. One of the main goals behind WRF-SFIRE is to model this feedback. [11] [22]. The codebase grew out of a serial implementation called the Clark-Hall mesoscale atmospheric model

coupled. Since WRF was built with parallelism in mind, it made sense to expose this parallelism by adopting it [11].

A. Farguell, A. Cortes, T. Margalef, J.R. Miro and J. Mercader note that the atmosphere is represented as a 3D grid while the fire variables are represented as a 2D grid [69].

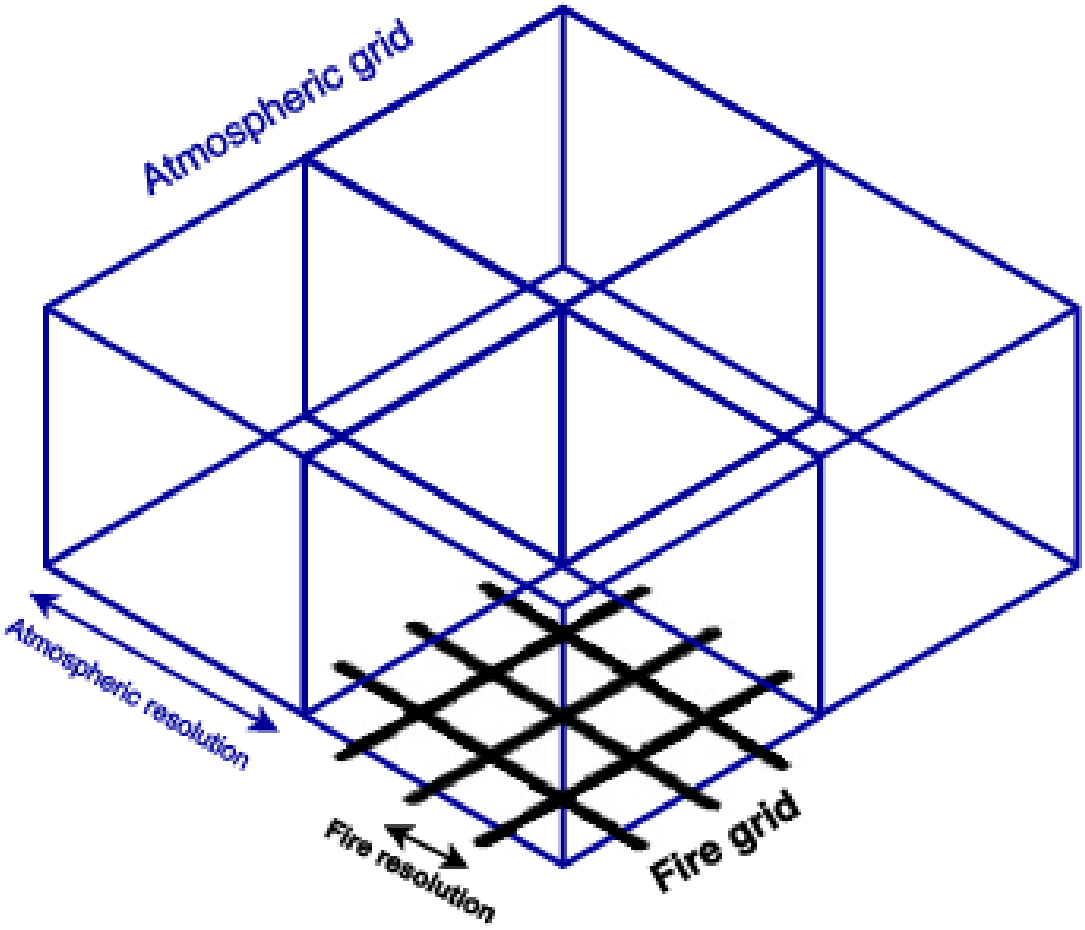

Fig. 14. $2x2$ atmospheric grid with fire mesh on the surface.
Source: [11]

G. Parallelism in WRF-SFIRE

The software architecture of WRF-SFIRE was designed to exploit different types of parallelism. In particular, the model domains, which are rectangular planes obtained by selecting regions of the earth and a map projection, are divided into patches to be distributed across nodes. These patches are then subdivided into tiles and worked on in parallel [1]. The decomposition of the model domain into patches is a coarse-grained model of parallelism where the data and work are divided into large partitions, called patches, for each node to work on. Subdividing those patches into tiles to work in parallel in the same

address space leverages data parallelism which would typically use something like OpenMP. WRF-SFIRE uses halo exchanges to efficiently communicate data between MPI processes. Developers can leverage the WRF MPI system by defining these halo exchanges in a registry file. The registry file, in this case, would generate the necessary code for WRF to communicate over MPI [12].

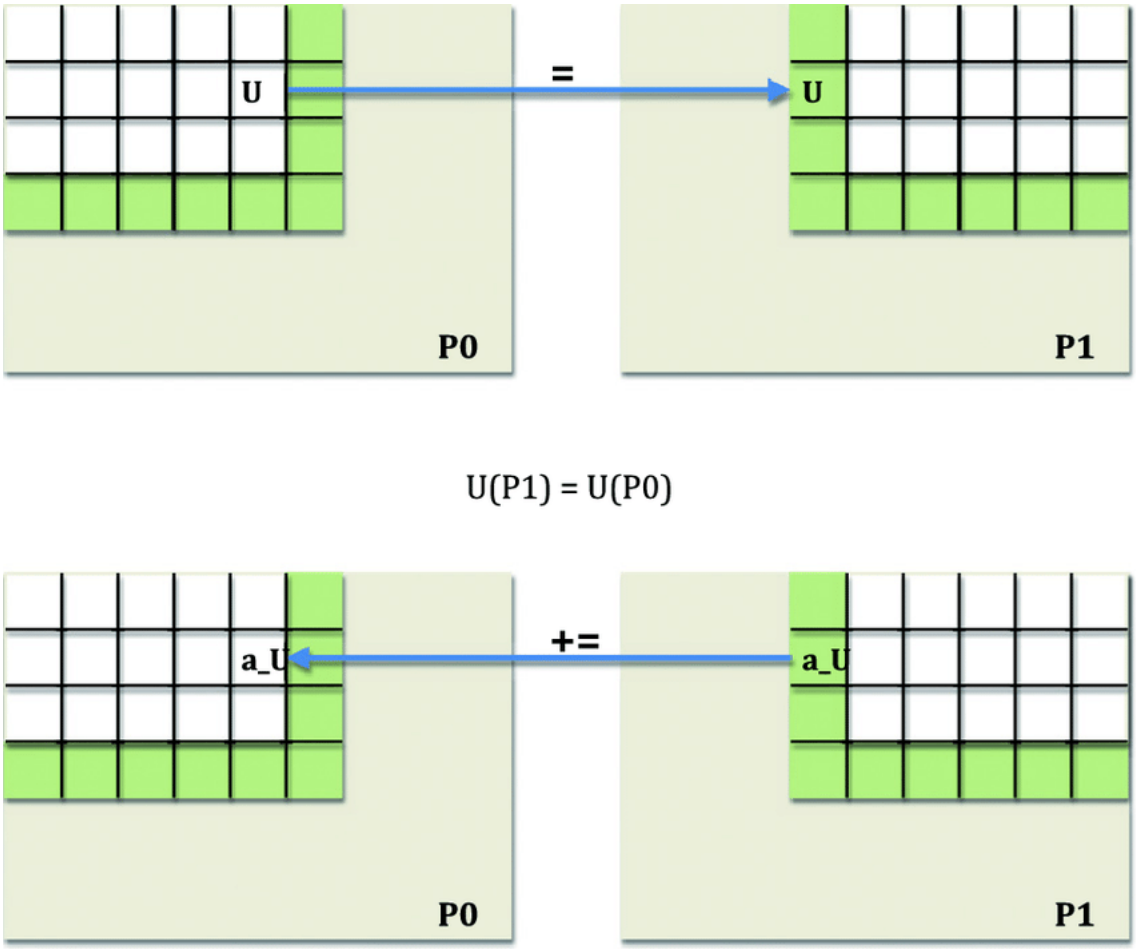

Fig. 15. A Halo Exchange between two neighboring processors.
Source: [12]

Each grid cell has a halo region that consists of a few rows on each of the 4 sides. This halo region is a contiguous local block of memory used to pass grid data from that particular grid cell to its neighboring grid cells over MPI [12].

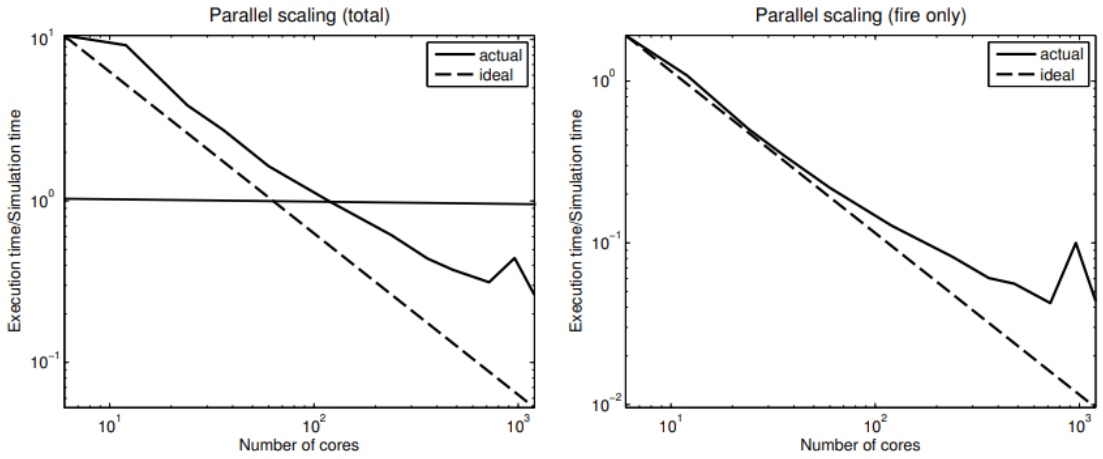

Fig. 16. Parallel performance of WRF-SFIRE as cores scale.
Source: [13]

G. Jordanov et al. [13] used data obtained from GIS, satellite imagery, and standard atmospheric data sources to simulate a fire in Harmanli, Bulgaria. The authors found that WRF-SFIRE runs slightly faster than real time in this case when ran on a cluster. They used three domains in the simulation, an outer domain with $250m$ resolution, an inner domain with $50m$ resolution and the fire mesh at $5m$ resolution coupled with the inner domain. They utilized a cluster with 120 cores for their simulations. Figure 16 shows their performance increased as core counts increased.

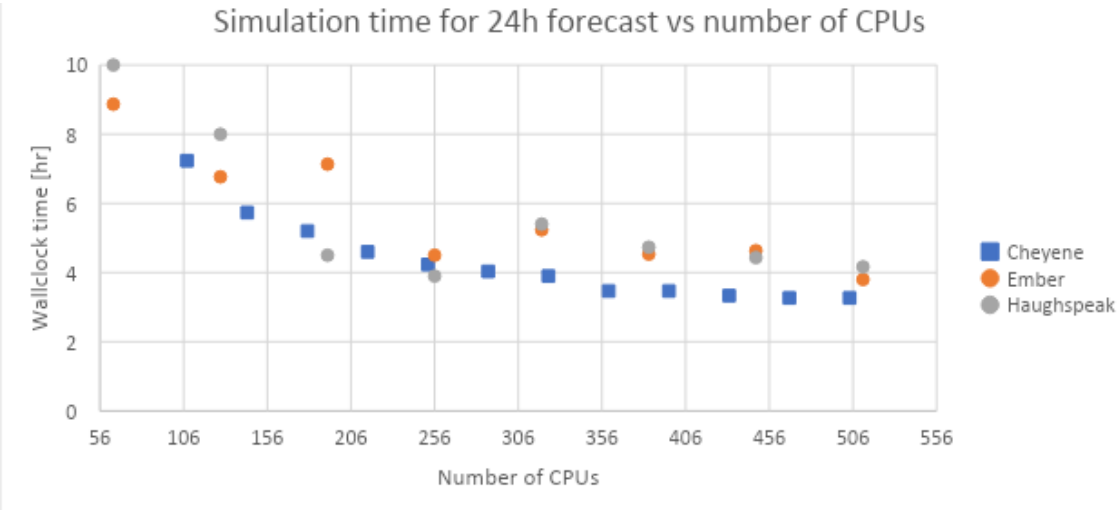

Fig. 17. Benchmarks of WRF-SFIRE run on the Cheyene, Haughspeak and Ember clusters .
Source: Kochanski, personal communication

The benchmarks in Figure 17 from Adam K. Kochanski and the OpenWFM team [10] compare the performance of WRF-SFIRE on three different clusters. The overall trend shows that MPI has provided a coarse grained level of parallelism that has allowed the coupled model to scale on many cores. However, J.

Michalakes and M. Vachharajani [59] showed that they were able to increase the overall execution speed by using GPUs to parallelize the fine-grained level of parallelism found in WRF. Currently, WRF-SFIRE uses OpenMP to exploit the fine-grained parallelism, particularly when working on tiles of a model domain patch in parallel. GPUs provide significantly more cores that can be used to do more of this work in parallel.

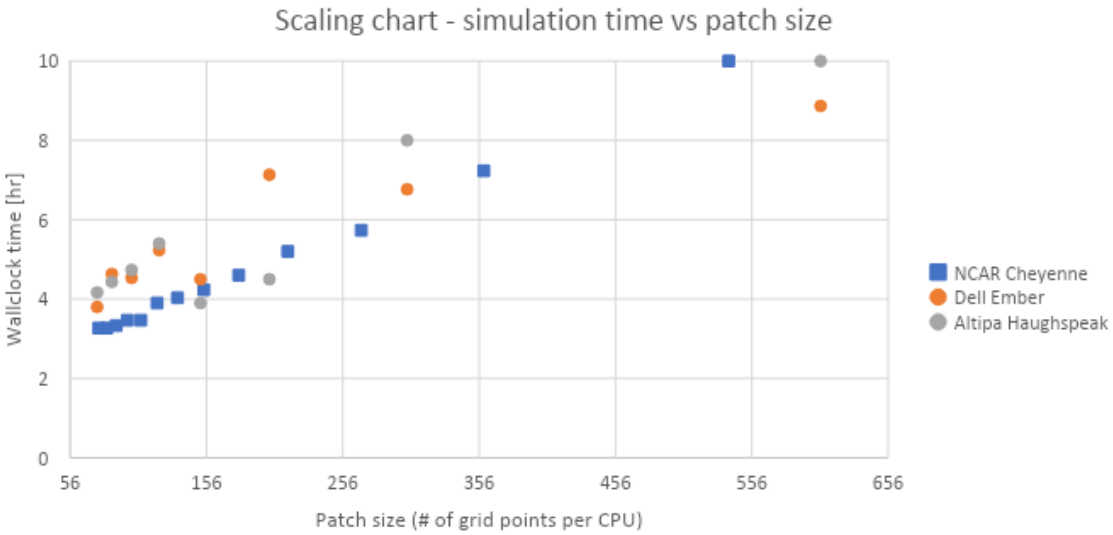

Fig. 18. Simulation time vs patch size.
Source: Kochanski, personal communication

Figure 18 shows that as the patch size increases, more work is given to each CPU and the execution time increases. Ideally, GPUs would ease the burden here because that work would be distributed to more cores in parallel.

V. GPU ACCELERATING WRF-SFIRE

A. Technical Approach

The following steps were used for the research process on the testing system. First a baseline benchmark was obtained to determine the normal execution time. Then the following steps were performed.

1. Profile the running code to find where the program spends the most time.
2. Utilize the Nvidia HPC SDK and tools to try to improve WRF-SFIRE's performance.
3. Profile the GPU sections of code to see how well the GPU is being utilized.
4. Compare the results to the baseline.
5. Repeat the above process.

B. First Profiling

The first major step was to setup the testing machine to compile WRF-SFIRE normally. This involved cherry picking fixes from the upstream WRF repository via git as well as compiling libraries such as NETCDF from source. Once it was successfully running using the GNU Fortran compiler, the program was executed and profiled using Perf and Oprofile to determine where the program was spending the most time. The top three results are as follows:

Function	percentage of time
$module\_first\_rk\_step\_part1\_first\_rk\_step\_part1$	49.90%
$solve\_em$:	33.10%
$module\_fr\_sfire\_driver\_wrf\_sfire\_driver\_em\_step\_$	3.43%

Table 4: Perf Results for nvfortran compiled WRF-SFIRE with MPI

The goal was to offload some of the computation in these hotspots to the GPU. The joint PGI and Nvidia HPC SDK was setup on the machine. The necessary libraries were compiled using the PGI compiler and WRF-SFIRE was successfully compiled.

C. The Game of Life

The modules that did the most work according to Table 4 were doing work on the WRF atmospheric grid which represented the model domain. According to the WRF-SFIRE wiki, this domain grid was a three dimensional, logically rectilinear grid. Other sources point out that the Advanced Research WRF (ARW) utilized a staggered Arakawa C-grid [70]. The source code showed that the grid was the domain and it contained many multidimensional arrays that represented the WRF-SFIRE domain. A smaller but similar problem was presented in order to understand the potential difficulties with using a large data structure. A Fortran implementation of The Game of Life [71], which was published on a website under a GNU Free Documentation License, was used. The implementation was incrementally modified to offload the computation to the GPU using the OpenMP syntax.

According to M. Gardner in the October 1970 issue of Scientific American [72], the Game of Life is a zero-player cellular automaton game created by John Horton Conway that contains the power of a universal Turing machine. This means that anything that is possible to compute can be computed in this game. It is made up of a two-dimensional matrix of cells. Each cell contains one of two states, alive or dead. The next generation of cells is calculated by applying a set of rules to each cell in the current generation. The state of a cell depends on its horizontally, vertically and diagonally adjacent neighbors. The rules are as follows [73]:

1. Any live cell with fewer than two live neighbors dies from underpopulation.

2. Any live cell with more than three live neighbors dies from overpopulation.

3. Any live cell with two or three live neighbors remains unchanged in the next generation.

4. Any dead cell with exactly three live neighbors comes to life in the next generation.

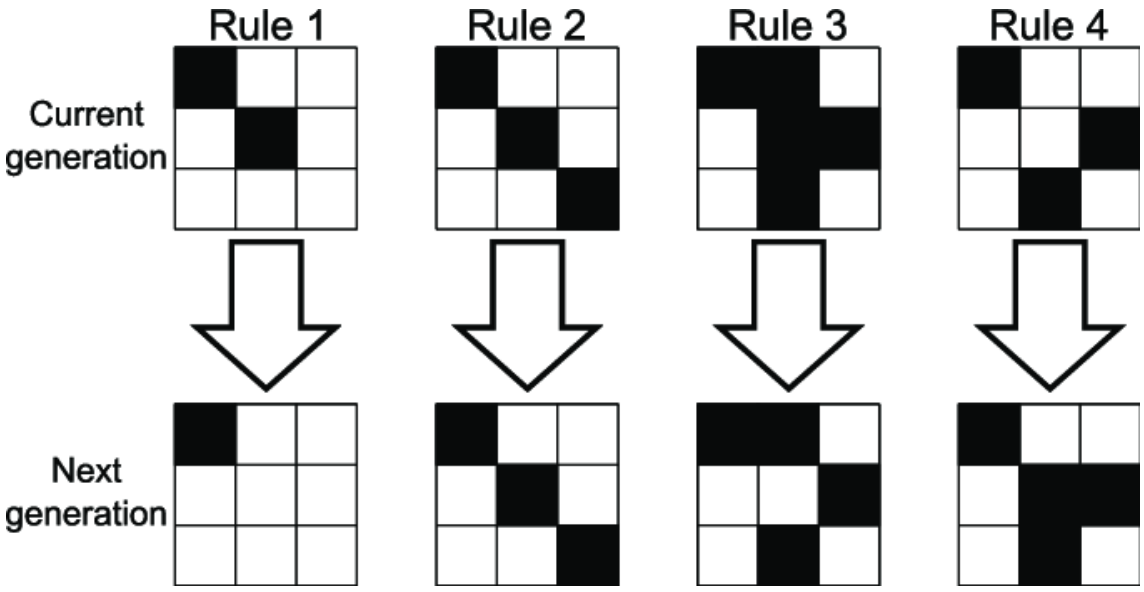

Fig. 19. Game of Life Rules.
Source: [14]

In this program, the main computation was applying the rules to a two dimensional grid of cells, as seen in Figure 20 and Figure 21. When this was performed, a copy of the cells grid was created so that changes to the original grid didn't affect the rest of the grid. Once the rules have been applied to every cell in the grid, the buffer becomes the new grid. One thing to note here was that a very expensive copy happened when creating the buffer and copying the cells into it. The first iteration of the the Game of Life program was to try to get anything working on the GPU. For a reference point, when compiled with gfortran, the configuration of 10 generations with a grid size of $10,000$ took 25.59 seconds without I/O to execute on the CPU.

```
SUBROUTINE Nextgen(cells)
   LOGICAL, INTENT(IN OUT):: cells(0:,0:)
   LOGICAL:: buffer(0:SIZE(cells, 1)-1, 0:SIZE(cells, 2)-1)
   INTEGER:: neighbours, i, j

   buffer = cells   ! Store current status
   DO j = 1, SIZE(cells, 2)-2
      DO i = 1, SIZE(cells, 1)-2
        if(buffer(i, j)) then
          neighbours = sum(count(buffer(i-1:i+1, j-1:j+1), 1)) - 1
        else
          neighbours = sum(count(buffer(i-1:i+1, j-1:j+1), 1))
        end if

        SELECT CASE(neighbours)
          CASE (0:1, 4:8)
             cells(i, j) = .FALSE.

          CASE (2)
             ! No change

          CASE (3)
             cells(i, j) = .TRUE.
        END SELECT

      END DO
   END DO
END SUBROUTINE Nextgen
```

Fig. 20. Subroutine that calculates the next generation

```
DO generation = 1, 10
   CALL Nextgen(cells)
   !CALL Drawgen(cells(1:gridsize, 1:gridsize), generation)
END DO
```

Fig. 21. Loop that iterates over each generation

OpenMP was applied to the outermost loop which distributed the work across thread blocks with a block size of $[1, 1, 1]$. With a grid size of $10,000$ and 10 generations, this configuration took 17 minutes and 47 seconds to execute. 99.8% of the GPU execution time was spent executing the kernel. The kernel took so long to execute because the inner loop was being executed sequentially due to the ineffective use of OpenMP directives.

Next, OpenMP was applied to the nested loops which iterated over each cell in the two dimensional grid, then performed the Game of Life computations and put the result into a buffer. Originally the performance suffered significantly compared to the normal execution using gfortran. Using a grid size of $10,000$ for 10 generations took the GPU 15 minutes and 52 seconds.

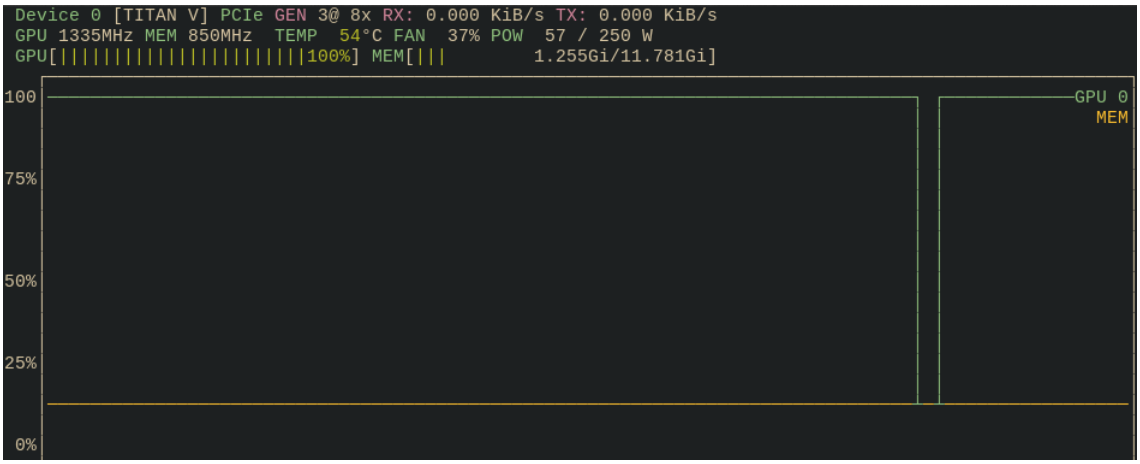

Fig. 22. GPU utilization graph using nvtop.

The GPU was at 100% utilization most of the time and would have occasional dips. A profiler from the Nvidia HPC SDK was used to profile and analyze the results. It appeared that the kernel being generated was allocating 1024 blocks, each with one thread. In other words, it was creating a grid size of $[1024, 1, 1]$ and a block size of $[1, 1, 1]$. In this case the current major bottleneck was that the generated kernel was executing the inner loops sequentially. It was using 55 registers for each block and the percentage of stalls from memory operations was 80.9%. This would mean that the buffer is being copied from host memory to the GPU memory and vice versa each time the next generation function is called. In fact, the output from compiling the code suggests that it was generating an implicit "ToFrom" mapping, which means that the data gets loaded into GPU memory and then returns the result to host memory.

1	Memcpy HtoD	0.301856s	1.664 µs	GPU 0	Stream 13
2	Memcpy HtoD	0.302018s	609.723 µs	GPU 0	Stream 13
3	Memcpy HtoD	0.303055s	606.299 µs	GPU 0	Stream 13
4	Memset	0.325077s	3.744 µs	GPU 0	Stream 7
5	Memcpy HtoD (Pageable)	0.325101s	1.216 µs	GPU 0	Stream 7
6	nvkernel_life_2d_nextgen_F1L62_1_	0.325458s	1.038 s	GPU 0	Stream 14
7	Memcpy DtoH	1.36378s	593.788 µs	GPU 0	Stream 13
8	Memcpy DtoH	1.36438s	593.595 µs	GPU 0	Stream 13
9	Memcpy HtoD	1.37082s	1.280 µs	GPU 0	Stream 13
10	Memcpy HtoD	1.3711s	605.755 µs	GPU 0	Stream 13
11	Memcpy HtoD	1.37171s	608.284 µs	GPU 0	Stream 13
12	nvkernel_life_2d_nextgen_F1L62_1_	1.37236s	1.076 s	GPU 0	Stream 14
13	Memcpy DtoH	2.44811s	593.276 µs	GPU 0	Stream 13
14	Memcpy DtoH	2.44871s	593.563 µs	GPU 0	Stream 13

Fig. 23. Nvidia Profiler.

Figure 23 confirmed that the data from host memory was being copied to device memory, then the kernel was executed and the results were copied back to the host for each generation that was calculated. However, the current major bottleneck is that the generated kernel was executing the inner loops sequentially.

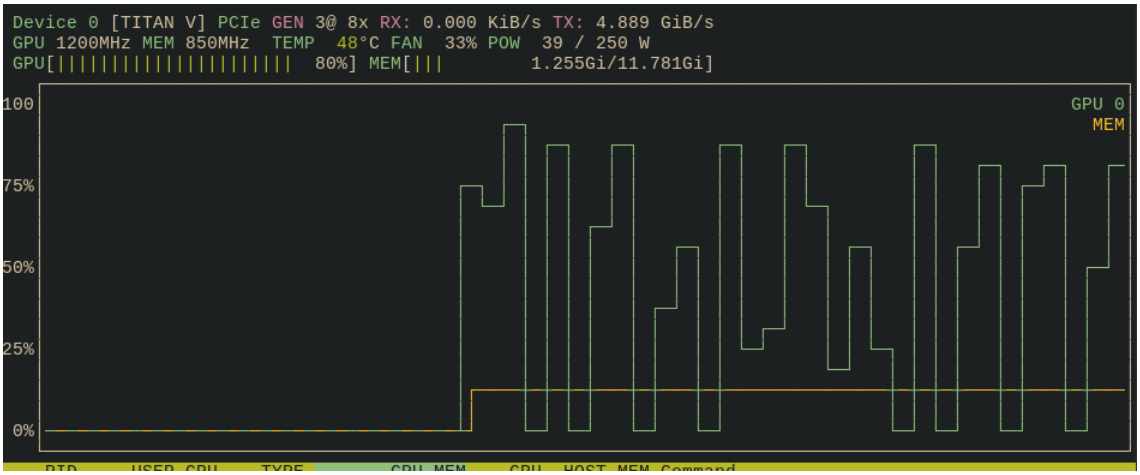

Fig. 24. GPU utilization graph using nvtop.

The next iterations of the Game of Life would incorporate the lessons from the other iterations. By being more explicit about how the GPU should distributed the workload, the allocated resources were more effectively utilized. This experiment did 10 generations with a grid size of $10,000$ cells. This time the GPU accelerated execution took 7.2 seconds. The profiler output showed that the compiler was allocating a grid size of $[1, 1, 1]$ and a block size of $[128, 1, 1]$. In other words, it was allocating one block with 128 thread processors.

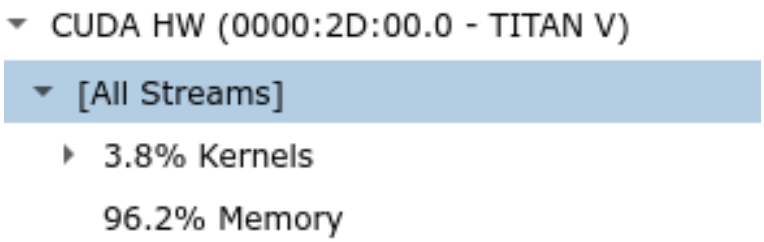

Fig. 25. Profiler Percentages

However, as seen in Figure 25 data was still being copied from host to device and vice versa after each generation. The solution to optimizing the data transfers was to load the data into GPU memory and keep it there until the end of the program. Another slow-down that occurred was that the cells array was being copied into the buffer by the CPU in host memory. This operation could be moved to the GPU.

Fig. 26. GPU utilization graph using nvtop.

For the final iteration of the program, the buffer was explicitly allocated in GPU memory and the cells array with its initial configuration was copied into GPU memory at the beginning of the program. OpenMP was used to tell the GPU to used the buffers that were already put into memory. It would then offload the copying of the cells array into the buffer all on the GPU. The compiler generated a second kernel for this. After profiling the program, it was allocated one block with 128 thread processors. With the same parameters, namely 10 generations with a 10000 cell grid, the program executes in 0.95 seconds. It has a maximum theoretical warp occupancy of 75% and the actual warp occupancy that was achieved was 73%. This means that the allocated resources are being used effectively.

The results are summarized in the table below.

Time	Optimizations
25.59 seconds	Baseline execution on the CPU without I/O
17 minutes 47 seconds	Distribute outer loop across thread blocks.
15 minutes 52 seconds	Distribute outer loop across thread blocks. And markup inner loop but it contains a scalar dependency.
7.2 seconds	Distribute outer loop across thread blocks. Distribute inner loop across threads.
0.95 seconds	Distribute outer loop across thread blocks. Distribute inner loop across threads. Optimize memory by allocating the buffer directly in the GPU memory and copying the initial state once into GPU memory. Copy the results out of GPU memory once the program has finished.

Table 5: Game of Life results with baseline CPU execution time of 25.59 seconds, grid size of 10000 and 10 generations

```
SUBROUTINE Nextgen(cells)
   LOGICAL, INTENT(IN OUT):: cells(0:,0:)
   !LOGICAL:: buffer(0:SIZE(cells, 1)-1, 0:SIZE(cells, 2)-1)
   INTEGER:: neighbours, i, j

   !buffer = cells   ! Store current status
      !$omp target teams loop collapse(1) map(to:cells(:,:)) map(alloc:buffer)
   DO j = 1, SIZE(cells, 2)-2
   !$omp loop bind(parallel)
      DO i = 1, SIZE(cells, 1)-2
      buffer(j, i) = cells(j, i)

      end do
      end do
  !$omp target teams loop collapse(1) map(to:cells(:,:)) map(to:buffer(:,:))
   DO j = 1, SIZE(cells, 2)-2
     !$omp loop bind(parallel)
      DO i = 1, SIZE(cells, 1)-2
        if(buffer(i, j)) then
          neighbours = sum(count(buffer(i-1:i+1, j-1:j+1), 1)) - 1
        else
          neighbours = sum(count(buffer(i-1:i+1, j-1:j+1), 1))
        end if

        SELECT CASE(neighbours)
          CASE (0:1, 4:8)
             cells(i, j) = .FALSE.

          CASE (2)
             ! No change

          CASE (3)
             cells(i, j) = .TRUE.
        END SELECT

      END DO
   END DO
END SUBROUTINE Nextgen
```

Fig. 27. Nextgen routine with GPU directives

```
!$omp target enter data map(to:cells(:,:)) !$omp map(alloc:buffer(0:SIZE(cells, 1)-1, 0:SIZE(cells,2)-1))
DO generation = 1, 100
   CALL Nextgen(cells)
   !CALL Drawgen(cells(1:gridsize, 1:gridsize), generation)
END DO
!$omp target exit data map(from:cells) map(delete:buffer)
```

Fig. 28. Memory optimizations

Figure 27 and Figure 28 show the code changes for the final optimizations.

D. Takeaways From Game of Life

There are a few common pitfalls that were encountered from working with the Game of Life. First, not effectively utilizing the resources of the GPU, perhaps by using the OpenMP or OpenACC macros, leads to starvation. Performance increases from utilizing the GPU relies on high throughput, which requires maximizing the the resources that the GPU has to offer. Second, it is beneficial to copy large pieces of data and keep them around in GPU memory for other kernels to use. Data transfers are expensive and can significantly slow down the program when done repeatedly.

E. Applying OpenACC to WRF-SFIRE

The Game of Life example above showed that data transfers and poor allocation of resources can become a performance bottleneck. One of the main concerns when accelerating WRF-SFIRE was the potential large data transfers between the host and device. Throughout the code, a data structure called the Domain, often called a grid, was used to encapsulate all of the information needed to model the physical environment and run the simulation. A pointer to this grid or pointers to the individual arrays in the grid were passed around the codebase often with indices used to index into the various arrays. WRF-SFIRE added another important data structure for dealing with the fire spread model called $fire\_params$, which contained items initialized from the WRF grid such as fire winds, terrain height, spread coefficients and fuel moisture. Constantly copying these arrays in and out of device memory could be expensive.

The approach with the following experiments was to try to accelerate portions of the code and determine if it made the execution faster. If it did not make the execution faster, then there is no sense in keeping it. The experiments were ran independently of each other to be sure of which change was affecting the execution time.

Simulation Minutes	1	5	10	20
PGI DMPAR Real Times	$0m14.451s$	$1m12.230s$	$2m26.746s$	$4m47.872s$

Table 6: NVFortran $fireflux\_small$ executions times without GPU acceleration with six MPI processes.

Table 6 shows the execution times of the $fireflux_small$ test case with six MPI processes and no GPU acceleration. These times are used as baseline to determine if the overall execution time is better or worse when offloading computation to the GPU.

1. solve_em

According to the profiling information from Table 4, a large portion of the time spent in the application occurs in *solve_em*. Although this subroutine was from WRF and was not specific to WRF-

SFIRE, it was worth starting there. At this point, the goal was to get anything to run on the GPU. The *solve_em* subroutine iterated over all of the tiles in the grid and performed all of the computations. The idea here was to create a CUDA grid for each tile. In a multi-gpu environment, these grids could be distributed to separate GPUs. The first call in this subroutine was to the *advance_uv* subroutine. At first glance, *advance_uv* looked promising because there were many loops and nested loops that were performing mathematical operations. The local arrays, *mudf_xy*, *dpxy*, and *dpn*, could be allocated directly onto the GPU instead of being allocated on the host and then copied into the GPU. The rest of the parameters needed to be copied to the GPU or utilized via CUDA unified memory. Most of the inner loops could in fact be ran in parallel on the GPU as seen in Figure 29.

```
!$acc loop gang vector(128) collapse(2)
 DO k = k_start, k_end
 DO i = i_start_u_tend, i_end_u_tend
   u(i, k,j) = u(i,k,j) + dts*ru_tend(i,k,j)
 ENDDO
 ENDDO
!$acc loop gang vector(128) collapse(2)
 DO i = i_start_up, i_end_up
   MUDF_XY(i)= -emdiv*dx*(MUDF(i, j)-MUDF(i-1,j))/msfuy(i,j)
 ENDDO
!$acc loop gang vector(128) collapse(2)
 DO k = k_start, k_end
 DO i = i_start_up, i_end_up
!    Comments on map scale factors:
!    x pressure gradient: ADT eqn 44, penultimate term on RHS
!       = -(mx/my)*(mu/rho)*partial dp/dx
!    [i.e., first rho->mu; 2nd still rho; alpha = 1/rho]
!    Klemp et al. splits into 2 terms:
!       mu alpha partial dp/dx + partial dp/dnu * partial dphi/dx
!    then into 4 terms:
!       mu alpha partial dp'/dx+nu mu alpha' partial dmubar/dx +
!     + mu partial dphi/dx + partial dphi'/dx * (partial dp'/dnu - mu')
!
!    first 3 terms:
!    ph, alt, p, al, pb not coupled
!    since we want tendency to fit ADT eqn 44 (coupled) we need to
!    multiply by (mx/my):
!
   dpxy(i, k)= (msfux(i,j)/msfuy(i,j))*.5*rdx*(c1h(k)*muu(i,j)+c2h(k))*(          &
     ((ph (i, k+1,j)-ph (i-1,k+1,j))+(ph (i,k,j)-ph (i-1,k,j)))  &
    +(alt(i, k  ,j)+alt(i-1,k  ,j))*(p  (i,k,j)-p  (i-1,k,j))  &
    +(al (i, k  ,j)+al (i-1,k  ,j))*(pb (i,k,j)-pb (i-1,k,j)) )
 ENDDO
 ENDDO
```

Fig. 29. Shows a few of the inner loop found in *advance_uv*. In particular, lines 805 to 838 in *dyn_emmodule_small_step_em.F*.

However, the local variables $mudf\_xy$, $dpxy$, and dpn prevent the parallelization of the main outermost loop on line 804 of $dyn\_em module\_small_s tep\_em.F$. These variables have a data dependency because they are used in other nested loops. This forces the outermost loop to be run sequentially. In fact, the compiler recommends that this loop is scheduled on the host. Similarly, these same variables prevent the parallelization of the loop labeled $v\_outer\_j\_loop$ at line 844 of $dyn\_em module\_small_s tep\_em.F$ for the same reason. The result is a significant increase in execution time because these loops must be executed sequentially on the GPU. GPU cores, however, have a slower clockrate than cores found in CPUs.

The next subroutine that is called from $solve\_em$ and at first glance is a promising candidate is $advance\_mu\_t$. Similar to $advance\_uv$, this subroutine contains a lot mathematical calculations and no other function calls. Unfortunately, the main loop in this subroutine has a data dependency on the $dvdxi$ array, so the loop must be run sequentially. Similarly, another major outerloop must be run sequentially due to a data dependency on the $wdtn$ array. This caused the execution time to be 7 seconds to simulate one simulation second on the $fireflux\_small$ test case with one MPI process. It gets even worse when adding more MPI processes. It takes 17.8 seconds to simulate one simulation second on the $fireflux\_small$ test case with six MPI processes.

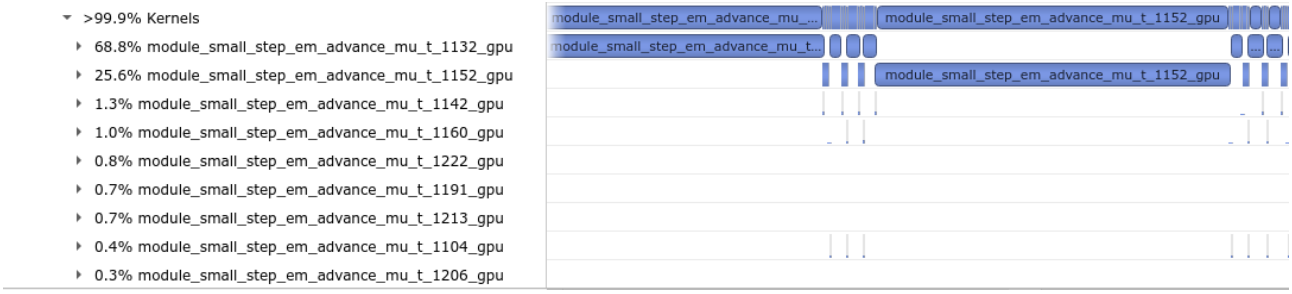

Fig. 30. Visual output from kernel executions of $advance\_mu\_t$ from Nvidia Nsight profiler with six MPI process.

According the Nvidia profilers (Figure 31), 88% of CUDA execution time is spent executing kernels. Of this 88%, about 60% of the time is spent executing one kernel, namely $module\_small\_step\_em\_advance\_uv\_884\_gpu$. This kernel is launched 1550 times and takes about 6.5 milliseconds to execute, as opposed to the next most time consuming kernel, which takes 71 microseconds to execute. The profiler reports that this kernel is being launched with one grid and one block and has a very low warp occupancy percentage, which is expected due to the sequential execution. Therefore, due to the data dependencies found in this function, it is not worth executing it on the GPU in its current form.

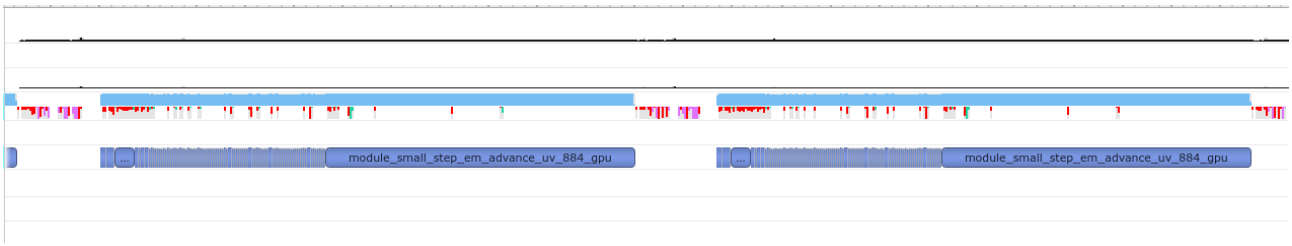

Fig. 31. Visual output from kernel executions of *advance_uv* from Nvidia Nsight profiler with one MPI process.

The Nividia profiler visual snapshot from Figure 30 shows that there are eight kernels, two of which take up most of the execution time. The data dependencies force these kernels to run sequentially and cannot utilize the resources of the GPU because it must be run on a grid with dimensions $< 1, 1, 1 >$.

My next attempt was to accelerate what is reasonable from *advance_uv* as well as other functions that are called from *solve_em*. The goal here was to execute each tile as a grid and call the functions in the loop found in *solve_em* as kernel routines. The difficulty here is that the subroutines were not explicitly designed to be kernel routines. The next subroutine called after *advance_uv* is $pxft$. When marking the $pxft$ subroutine as a kernel routine, OpenACC required that the highest level of parallelism allowed was explicitly defined for that routine. For example, the loop found in *solve_em* will be executed as grids, also known as gangs. This means that all of the subroutines launched within that kernel region must be either workers or vectors. Similarly, if a call to another subroutine was from a routine that was being executed with worker level parallelism, the current routine was limited to launching that subroutine with the vector level or parallelism. All other subroutines that get launched from a routine with vector level parallelism must be run sequentially. This means that subroutines can nest at most two functions calls to be run with some lower level of parallelism from a grid kernel region. Ideally these kernel routines should be designed to perform at the level of parallelism intended.

However, the $pxft$ subroutine found in $dyn\_em/module\_polarfft.F$, contains many nested subroutine calls, some of which cannot be accelerated, such as wrf_error_fatal for example. Many of the nested subroutines are also not compute intensive and would not effectively utilize the GPU. This means that $pxft$ must be launched on the host. In doing so, OpenACC must copy out the necessary data to the host, execute it on the host and then copy the results back into the GPU. It is not feasible to accelerate *solve_em* in this way.

2. module_fr_sfire_driver.F

This next section of code is the $sfire\_driver\_em$ subroutine which is the main driver that advances the fire model. According to the code, the fire model inputs wind data as well as other constant arrays such as fuel data, advances the model by a timestep and outputs the temperature and humidity tendencies [10]. The $sfire\_driver\_em$ subroutine instantiates the necessary data structures, determines if the fuel moisture model is necessary, performs the needed halo exchanges, executes the fire physics ($sfire\_driver\_phys$), communicates the necessary information via more halo exchanges and finally adds the fire emissions outputs for WRF to simulate in the atmospheric model. Therefore, the main WRF-SFIRE code that will run on a patch in each MPI process is the $sfire\_driver\_phys$ subroutine. There are many parameters for this subroutine, most of which are arrays from the grid data structure and the $fire\_params$. Once invoked, the patch is divided up into tiles and the tiles are executed in parallel via OpenMP.

The ideal execution here is to copy all of the data that is needed to work on the tile into GPU memory at once, perform all of the compute work on the GPU and copy the results back. My starting point here is the $sfire\_driver\_phys$ subroutine found in $phys/module\_fr\_sfire\_driver.F$. This subroutine calls the main fire model subroutine called $sfire\_model$ found in $phys/module\_fr\_sfire\_model.F$. The $sfire\_model$ subroutine has promising loops and many function calls. OpenACC directives were applied to the loops in the $sfire\_model$ subroutine and the $heat\_fluxes$ subroutine which was called by $sfire\_model$. However, rather than limit the level of parallelism by trying to make the $heat\_fluxes$ subroutine a kernel routine, it was inlined. As a reference point, the execution time prior to these modifications was 20.972 seconds for one simulation minute using six MPI processes.

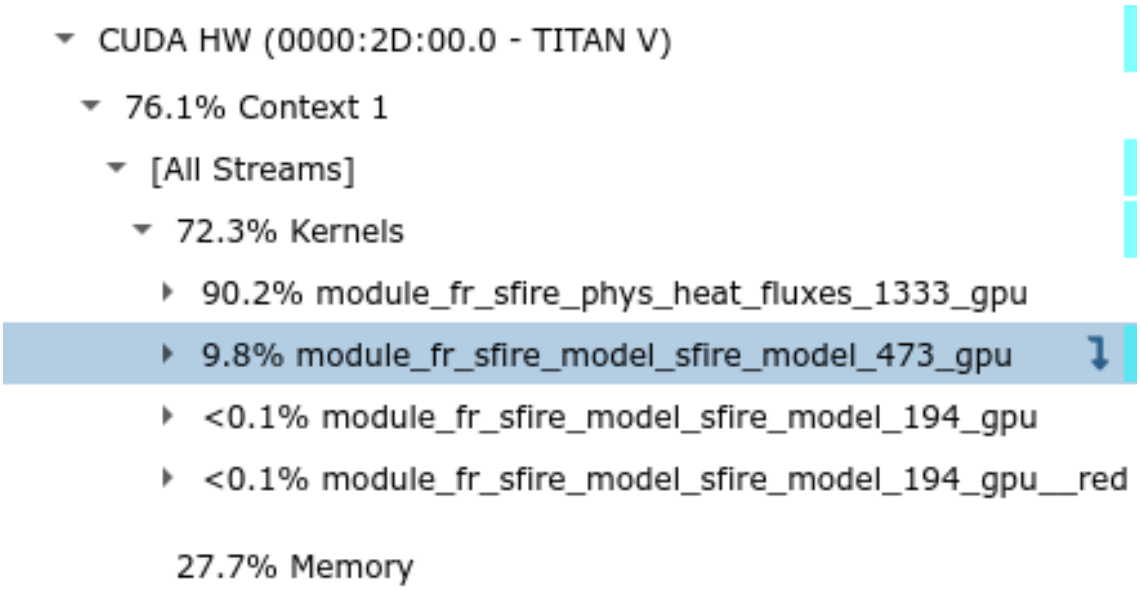

Fig. 32. Visual output from kernel executions of *heat_fluxes* and *sfire_model* from Nvidia Nsight profiler with four MPI process.

After the modifications, the execution time increased to 33.764 seconds. In the loops that OpenACC directives were applied, there were no data dependencies which allowed for the parallelization of the loops. We can see from Figure 32 that 72.3% of the execution time was spent mostly in the top two kernels. 27.7% of the time was spent copying data from the host to the device for execution and then copying the results back.

```
!*** executable
latent = present(grnqft)
!$acc parallel loop gang, vector collapse(2)
do j = jfts, jfte
   do i = ifts, ifte
        dmass =                    &     ! ground fuel dry mass burnt th
            fgip(i, j)              &     ! init mass from fuel model nc
            * fuel_frac_burnt(i, j)       ! fraction burned this call
        bmst     = fp%fmc_g(i, j)/(1.+fp%fmc_g(i,j))
        grnhft(i, j) = (dmass/dt)*(1.-bmst)*cmbcnst         ! J/m^2/sec
       if(latent)grnqft(i, j) = (bmst+(1.-bmst)*.56)*(dmass/dt)*xlv  !
        ! bmst = relative water contents; 0.56 = est. ratio of water from
        ! xlv = nominal specific latent heat of water J/kg (dependence or
        ! xlv is  defined in module_model_constants
   enddo
enddo
!$acc end parallel
```

Fig. 33. OpenACC directives on main loop inside of *heat_fluxes* subroutine.

Many OpenACC directive configurations were used and profiled, and the best resulting configuration is shown in Figure 33. This configuration works best for this code region but according to the profiling tools, this only allows for a theoretical warp occupancy of 37%. There just isn't enough work in the loop to saturate the allocated resources. The second kernel, $module\_fr\_sfire\_model\_sfire\_model\_473\_gpu$, has a higher potential with a theoretical warp occupancy of about 62% but the achieved warp occupancy is about 33%. According the profiler, this usually means that there is high warp scheduling overhead and that the workloads may be imbalanced.

The next subroutine considered was $interpolate\_wind2fire\_height$. At first glance, it looked like a good routine to try to accelerate because it contained many calculations and functions calls to $course$ and $interpolate\_h$ could be inlined. It does, however, contain a $goto$ and a subroutine call to $crash$. For this experiment, both of those lines were commented out. In this subroutine, there was a scalar dependency on the variable ht. This meant that it was a data dependency that was carried over a single memory location [74].

```
!$acc kernels
 loop_j: do j = jftst, jftet
   call coarse(j, jr,-2,jcb,wjcb)    ! get interpolation coeffic
   call coarse(j, jr,ir,jcm,wjcm)    ! get interpolation coeffic
   loop_i: do i = iftst, iftet
     call coarse(i, ir,-2,icb,wicb)    ! get interpolation coeff
     call coarse(i, ir,ir,icm,wicm)    ! get interpolation coeff
     z0 = fz0(i, j)                    ! roughness length
     wh = fwh(i, j)                    ! wind height

     if( wh > z0 .and. z0 > 0)then

     ht_last = z0                     ! initialize starting heig
     loop_k: do k = kds, kdmax         ! search for layer k such
       ! interpolate height from atmospheric cell midpoints
       ht = interpolate_h(its-1, ite+1,kds,kde,jts-1,jte+1,icm,k
       if(.not. ht < wh) exit loop_k  ! found layer k this point
       ht_last = ht
     enddo loop_k
     !UNCOMMENT
    !if(k .gt. kdmax) then
       !call crash('interpolate_wind2fire_height: fire wind heig
       !goto 91  ! run out of vertical levels, this must be wron
     !endif
     kmin = min(k, kmin)
     kmax = max(k, kmax)
     ! found layer k, ht_last < wh <= ht
     logz0 = log(z0)
     logwh = log(wh)
     loght_last = log(ht_last)
     loght = log(ht)
```

Fig. 34. Scalar dependency from *interpolate_wind2fire_height*.

In the above code, *ht* must be calculated before execution can continue. The loop *loop_k* will only continue based on the value of *ht* and *loght* must be set from *ht*. For this reason, the outer loop, *loop_j* and a nested loop, *loop_i* must be run sequentially. In fact, *loop_i* also has a scalar dependency for *kmin* and *kmax*. It appears that the biggest block for parallelizing WRF-SFIRE is data dependencies found in the compute-intensive loops.

F. Limitations

There were a few limitations, both with the methodology and WRF-SFIRE itself. First, I only had one GPU at my disposal which turned out to be a problem because multiple MPI processes would compete for GPU resources. In some cases, the execution time was reduced by reducing the number of MPI processes used. Second, compilation of WRF-SFIRE must be done serially. This was discovered quickly through experimentation since compilation would simply fail if it was multithreaded. This also means that making code changes and testing the results can take a really long time. With WRF-SFIRE itself, it seems that the number one problem that I faced was data dependencies. However, there are other problems such as deeply nested functions calls, functions calls that write to standard output or a file, and goto's appear in regions that are good candidates for parallel execution on the GPU. The final limitation that I'd like to mention occurs, when using OpenACC with WRF-SFIRE. Many of the variable names in the codebase end with a numeric character such as r_0 or $t2$. However, the compiler threw syntax errors when looking at those OpenACC directives. In those instances, I used my editor to search and replace those characters with words. For example, $t2$ became t_two.

G. Conclusions and Future Work

In this research, I was able to setup a GPU enabled environment for WRF-SFIRE and offload portions of the computation to the GPU. However, due to the complexity from the limitations of the code itself as described insubsection E. and subsection F. , I was unable to accelerate the execution in any meaningful way. GPUs perform best on compute intensive kernels that can be done in parallel. However, many of the compute intensive regions of WRF-SFIRE contained data dependencies that prevented effective parallel execution and resulted in an increase in execution time. This correlated with the behavior found when trying to accelerate The Game of Life. In both, WRF-SFIRE and The Game of Life, ineffective usage of GPU resources, especially when doing a lot of work sequentially, significantly slowed down the execution time. In the literature, many of the algorithms that were being accelerated to run on GPUs had to be rewritten in order to fully utilize the resources of a GPU. In my research, OpenACC was used to accelerate portions of the code without significant modifications to the original code. However, the regions of code that contained data dependencies need to be rewritten to avoid them if possible. This would require working with someone with expertise in the domain and who understands the mathematics and algorithms behind WRF-SFIRE.

It is also common in the codebase to use goto's and call functions within loops, particularly to debug functions that wrote to files or standard output. However, parallel regions found in CUDA kernels

cannot leave their respective parallel regions unless the call is to another kernel. In order to parallelize WRF-SFIRE compute intensive regions, the code should be rewritten to avoid data dependencies, goto's and subroutine calls that are not CUDA kernels as much as possible. When writing the CUDA kernels in the future, care should be taken to minimize both, host-to-device and device-to-host data transfers. I believe that if these regions are reorganized, they can be effectively executed at the fine-grained patch and tile level of WRF-SFIRE.

REFERENCES

[1] A. Z. Abualkishik, "A COMPARATIVE STUDY ON THE SOFTWARE ARCHITECTURE OF WRF AND OTHER NUMERICAL WEATHER PREDICTION MODELS," *Journal of Theoretical and Applied Information Technology*, vol. 96, p. 24, Dec. 2018.

[2] R. Busseuil, G. M. Almeida, L. Ost, S. Varyani, G. Sassatelli, and M. Robert, "Adaptation Strategies in Multiprocessors System on Chip," in *VLSI-SoC: Forward-Looking Trends in IC and Systems Design*, J. L. Ayala, D. Atienza Alonso, and R. Reis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 373, pp. 233–257. [Online]. Available: http://link.springer.com/10.1007/978-3-642-28566-0_10

[3] M. Ilg, J. Rogers, and M. Costello, "Projectile Monte-Carlo Trajectory Analysis Using a Graphics Processing Unit," in *AIAA Atmospheric Flight Mechanics Conference*. Portland, Oregon: American Institute of Aeronautics and Astronautics, Aug. 2011. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2011-6266

[4] "Parallel Programming for Multicore Machines Using OpenMP and MPI." [Online]. Available: https://ocw.mit.edu/courses/earth-atmospheric-and-planetary-sciences/12-950-parallel-programming-for-multicore-machines-using-openmp-and-mpi-january-iap-2010/

[5] "CUDA Refresher: Reviewing the Origins of GPU Computing," Apr. 2020. [Online]. Available: https://developer.nvidia.com/blog/cuda-refresher-reviewing-the-origins-of-gpu-computing/

[6] L. Durant, O. Giroux, M. Harris, and N. Stam, "Inside Volta: The World's Most Advanced Data Center GPU," May 2017. [Online]. Available: https://developer.nvidia.com/blog/inside-volta/

[7] P. Gupta, "CUDA Refresher: The CUDA Programming Model," Jun. 2020. [Online]. Available: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

[8] J. Murphy, "Accelerating Code with OpenACC and the NVIDIA Visual Profiler," Mar. 2016. [Online]. Available: https://www.microway.com/hpc-tech-tips/accelerating-code-with-openacc-and-nvidia-visual-profiler/

[9] "Unified Memory in CUDA 6," Nov. 2013. [Online]. Available: https://developer.nvidia.com/blog/unified-memory-in-cuda-6/

[10] "WRF-Fire." [Online]. Available: https://wiki.openwfm.org/wiki/WRF-Fire

[11] J. Mandel, J. D. Beezley, and A. K. Kochanski, "Coupled atmosphere-wildland fire modeling with WRF 3.3 and SFIRE 2011," *Geoscientific Model Development*, vol. 4, no. 3, pp. 591–610, Jul. 2011, number: 3. [Online]. Available: https://gmd.copernicus.org/articles/4/591/2011/

[12] X. Zhang, X.-Y. Huang, and N. Pan, "Development of the Upgraded Tangent Linear and Adjoint of the Weather Research and Forecasting (WRF) Model," *Journal of Atmospheric and Oceanic Technology*, vol. 30, no. 6, pp. 1180–1188, Jun. 2013. [Online]. Available: http://journals.ametsoc.org/doi/10.1175/JTECH-D-12-00213.1

[13] G. Jordanov, J. D. Beezley, N. Dobrinkova, A. K. Kochanski, J. Mandel, and B. Sousedík, "Simulation of the 2009 Harmanli Fire (Bulgaria)," in *Large-Scale Scientific Computing*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, I. Lirkov, S. Margenov, and J. Waśniewski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 7116, pp. 291–298. [Online]. Available: http://link.springer.com/10.1007/978-3-642-29843-1_33

[14] T. Hirose and T. Sawaragi, "Extended FRAM model based on cellular automaton to clarify complexity of socio-technical systems and improve their safety," *Safety Science*, vol. 123, p. 104556, Mar. 2020. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0925753519321678

[15] "Specification | OpenACC." [Online]. Available: https://www.openacc.org/specification

[16] J. Mendel, J. D. Beezley, J. L. Coen, and M. Kim, "Data assimilation for wildland fires," *IEEE Control Systems*, vol. 29, no. 3, pp. 47–65, Jun. 2009, number: 3. [Online]. Available: https://ieeexplore.ieee.org/document/4939311/

[17] A. Kochanski, M. Jenkins, J. Mandel, J. Beezley, and S. Krueger, "Real time simulation of 2007 Santa Ana fires," *Forest Ecology and Management*, vol. 294, pp. 136–149, Apr. 2013. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0378112712007372

[18] National Center for Atmospheric Research, "Weather Research and Forecasting Model," May 2021. [Online]. Available: https://www.mmm.ucar.edu/weather-research-and-forecasting-model

[19] J. L. Coen, M. Cameron, J. Michalakes, E. G. Patton, P. J. Riggan, and K. M. Yedinak, "WRF-Fire: Coupled Weather–Wildland Fire Modeling with the Weather Research and Forecasting Model," *Journal of Applied Meteorology and Climatology*, vol. 52, no. 1, pp. 16–38, Jan. 2013, number: 1. [Online]. Available: https://journals.ametsoc.org/view/journals/apme/52/1/jamc-d-12-023.1.xml

[20] A. K. Kochanski, M. A. Jenkins, J. Mandel, J. D. Beezley, C. B. Clements, and S. Krueger, "Evaluation of WRF-SFIRE performance with field observations from the FireFlux experiment," Atmospheric Sciences, preprint, Jan. 2013. [Online]. Available: https://gmd.copernicus.org/preprints/6/121/2013/gmdd-6-121-2013.pdf

[21] D. V. Mallia, A. K. Kochanski, K. E. Kelly, R. Whitaker, W. Xing, L. E. Mitchell, A. Jacques, A. Farguell, J. Mandel, P. Gaillardon, T. Becnel, and S. K. Krueger, "Evaluating Wildfire Smoke Transport Within a Coupled Fire-Atmosphere Model Using a High-Density Observation Network for an Episodic Smoke Event Along Utah's Wasatch Front," *Journal of Geophysical Research: Atmospheres*, vol. 125, no. 20, Oct. 2020. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1029/2020JD032712

[22] D. V. Mallia, A. K. Kochanski, S. P. Urbanski, J. Mandel, A. Farguell, and S. K. Krueger, "Incorporating a Canopy Parameterization within a Coupled Fire-Atmosphere Model to Improve a Smoke Simulation for a Prescribed Burn," *Atmosphere*, vol. 11, no. 8, p. 832, Aug. 2020. [Online]. Available: https://www.mdpi.com/2073-4433/11/8/832

[23] A. K. Kochanski, D. V. Mallia, M. G. Fearon, J. Mandel, A. H. Souri, and T. Brown, "Modeling Wildfire Smoke Feedback Mechanisms Using a Coupled Fire-Atmosphere Model With a Radiatively Active Aerosol Scheme," *Journal of Geophysical Research: Atmospheres*, vol. 124, no. 16, pp. 9099–9116, Aug. 2019. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1029/2019JD030558

[24] A. K. Kochanski, M. A. Jenkins, K. Yedinak, J. Mandel, J. Beezley, and B. Lamb, "Toward an integrated system for fire, smoke and air quality simulations," *International Journal of Wildland Fire*, vol. 25, no. 5, p. 534, 2016. [Online]. Available: http://www.publish.csiro.au/?paper=WF14074

[25] A. K. Kochanski, J. D. Beezley, J. Mandel, and C. B. Clements, "Air pollution forecasting by coupled atmosphere-fire model WRF and SFIRE with WRF-Chem," *arXiv:1304.7703 [physics]*, Apr. 2013, arXiv: 1304.7703. [Online]. Available: http://arxiv.org/abs/1304.7703

[26] V. Herr, A. K. Kochanski, V. V. Miller, R. McCrea, D. O'Brien, and J. Mandel, "A method for estimating the socioeconomic impact of Earth observations in wildland fire suppression decisions," *International Journal of Wildland Fire*, vol. 29, no. 3, p. 282, 2020. [Online]. Available: http://www.publish.csiro.au/?paper=WF18237

[27] National Center for Atmospheric Research, "WRF scaling and timing," May 2021. [Online]. Available: https://www2.cisl.ucar.edu/resources/wrf-scaling-and-timing

[28] J. Mielikainen, B. Huang, J. Wang, H.-L. Allen Huang, and M. D. Goldberg, "Compute unified device architecture (CUDA)-based parallelization of WRF Kessler cloud microphysics scheme," *Computers & Geosciences*, vol. 52, pp. 292–299, Mar. 2013. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0098300412003457

[29] M. Huang, J. Mielikainen, B. Huang, H. Chen, H.-L. A. Huang, and M. D. Goldberg, "Development of efficient GPU parallelization of WRF Yonsei University planetary boundary layer scheme," Nov. 2014. [Online]. Available: https://gmd.copernicus.org/preprints/7/8031/2014/gmdd-7-8031-2014.pdf

[30] M. Huang, B. Huang, Y.-L. Chang, J. Mielikainen, H.-L. A. Huang, and M. D. Goldberg, "Efficient Parallel GPU Design on WRF Five-Layer Thermal Diffusion Scheme," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 8, no. 5, pp. 2249–2259, May 2015, number: 5. [Online]. Available: http://ieeexplore.ieee.org/document/7110533/

[31] J. Mielikainen, B. Huang, H.-L. A. Huang, and M. D. Goldberg, "Improved GPU/CUDA Based Parallel Weather and Research Forecast (WRF) Single Moment 5-Class (WSM5) Cloud Microphysics," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 5, no. 4, pp. 1256–1265, Aug. 2012, number: 4. [Online]. Available: http://ieeexplore.ieee.org/document/6182591/

[32] J. Mielikainen, B. Huang, H.-L. A. Huang, M. D. Goldberg, and A. Mehta, "Speeding Up the Computation of WRF Double-Moment 6-Class Microphysics Scheme with GPU," *Journal of Atmospheric and Oceanic Technology*, vol. 30, no. 12, pp. 2896–2906, Dec. 2013, number: 12. [Online]. Available: http://journals.ametsoc.org/doi/10.1175/JTECH-D-12-00218.1

[33] R. Ridwan, A. I. Kistijantoro, M. Kudsy, and D. Gunawan, "Performance evaluation of hybrid parallel computing for WRF model with CUDA and OpenMP," in *2015 3rd International Conference on Information and Communication Technology (ICoICT)*. Nusa Dua, Bali, Indonesia: IEEE, May 2015, pp. 425–430. [Online]. Available: http://ieeexplore.ieee.org/document/7231463/

[34] M. Pharr and R. Fernando, Eds., *GPU gems 2: programming techniques for high-performance graphics and general-purpose computation*, ser. GPU gems. Upper Saddle River, NJ: Addison-Wesley, 2005, no. 2, oCLC: ocm57316708.

[35] "HPC SDK | NVIDIA," Mar. 2020. [Online]. Available: https://developer.nvidia.com/hpc-sdk

[36] M. Flynn, "Flynn's Taxonomy," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Boston, MA: Springer US, 2011, pp. 689–697. [Online]. Available: http://link.springer.com/10.1007/978-0-387-09766-4_2

[37] Y. Lin and V. Grover, "Using CUDA Warp-Level Primitives," Jan. 2018. [Online]. Available: https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/

[38] tim.lewis, "OpenMP FAQ." [Online]. Available: https://www.openmp.org/about/openmp-faq/

[39] K. Fürlinger and M. Gerndt, "Analyzing Overheads and Scalability Characteristics of OpenMP Applications," in *High Performance Computing for Computational Science - VECPAR 2006*, M. Dayé, J. M. L. M. Palma, A. L. G. A. Coutinho, E. Pacitti, and J. C. Lopes, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, vol. 4395, pp. 39–51. [Online]. Available: http://link.springer.com/10.1007/978-3-540-71351-7_4

[40] J. Neal, T. Fewtrell, and M. Trigg, "Parallelisation of storage cell flood models using OpenMP," *Environmental Modelling & Software*, vol. 24, no. 7, pp. 872–877, Jul. 2009. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1364815208002363

[41] P. Bates and A. De Roo, "A simple raster-based model for flood inundation simulation," *Journal of Hydrology*, vol. 236, no. 1-2, pp. 54–77, Sep. 2000. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S002216940000278X

[42] A. Afzal, Z. Ansari, and M. K. Ramis, "Parallelization of Numerical Conjugate Heat Transfer Analysis in Parallel Plate Channel Using OpenMP," *Arabian Journal for Science and Engineering*, vol. 45, no. 11, pp. 8981–8997, Nov. 2020. [Online]. Available: https://link.springer.com/10.1007/s13369-020-04640-1

[43] H. M. Bücker and X. S. Li, Eds., *2020 Proceedings of the SIAM Workshop on Combinatorial Scientific Computing*. Philadelphia, PA: Society for Industrial and Applied Mathematics, Jan. 2020. [Online]. Available: https://epubs.siam.org/doi/book/10.1137/1.9781611976229

[44] Open MPI documentation. [Online]. Available: https://www.open-mpi.org/doc/

[45] B. Armstrong, S. W. Kim, and R. Eigenmann, "Quantifying Differences between OpenMP and MPI Using a Large-Scale Application Suite," in *High Performance Computing*, G. Goos, J. Hartmanis, J. van Leeuwen, M. Valero, K. Joe, M. Kitsuregawa, and H. Tanaka, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, vol. 1940, pp. 482–493. [Online]. Available: http://link.springer.com/10.1007/3-540-39999-2_45

[46] V. D. Tran and L. Hluchy, "Parallelizing Flood Models with MPI: Approaches and Experiences," in *Computational Science - ICCS 2004*, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, M. Bubak, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, vol. 3036, pp. 425–428. [Online]. Available: http://link.springer.com/10.1007/978-3-540-24685-5_57

[47] F. Cappello and D. Etiemble, "MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks," in *ACM/IEEE SC 2000 Conference (SC'00)*. Dallas, TX, USA: IEEE, 2000, pp. 12–12. [Online]. Available: http://ieeexplore.ieee.org/document/1592725/

[48] "IBM PARALLEL SYSTEM SUPPORT PROGRAMS VERSION 2.2 AND RELATED ENHANCEMENTS." [Online]. Available: https://www.ibm.com/common/ssi/ShowDoc.wss?docURL=/common/ssi/rep_ca/7/897/ENUS296-257/index.html&lang=en_GB-ZZ-ZZ

[49] B. Yan and R. A. Regueiro, "Comparison between pure MPI and hybrid MPI-OpenMP parallelism for Discrete Element Method (DEM) of ellipsoidal and poly-ellipsoidal particles," *Computational Particle Mechanics*, vol. 6, no. 2, pp. 271–295, Apr. 2019. [Online]. Available: http://link.springer.com/10.1007/s40571-018-0213-8

[50] G. Chen, G. Li, S. Pei, and B. Wu, "Gpgpu supported cooperative acceleration in molecular dynamics," in *2009 13th International Conference on Computer Supported Cooperative Work in Design*, 2009, pp. 113–118.

[51] A. J. Kalyanapu, S. Shankar, E. R. Pardyjak, D. R. Judi, and S. J. Burian, "Assessment of GPU computational enhancement to a 2D flood model," *Environmental Modelling & Software*, vol. 26, no. 8, pp. 1009–1016, Aug. 2011. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1364815211000582

[52] R. Kelly, "GPU Computing for Atmospheric Modeling," *Computing in Science & Engineering*, vol. 12, no. 4, pp. 26–33, Jul. 2010. [Online]. Available: http://ieeexplore.ieee.org/document/5406490/

[53] P. E. Bieringer, A. J. Piña, D. M. Lorenzetti, H. J. J. Jonker, M. D. Sohn, A. J. Annunzio, and R. N. Fry, "A Graphics Processing Unit (GPU) Approach to Large Eddy Simulation (LES) for Transport and Contaminant Dispersion," *Atmosphere*, vol. 12, no. 7, p. 890, Jul. 2021. [Online]. Available: https://www.mdpi.com/2073-4433/12/7/890

[54] I. Demeshko, N. Maruyama, H. Tomita, and S. Matsuoka, "Multi-GPU Implementation of the NICAM Atmospheric Model," in *Euro-Par 2012: Parallel Processing Workshops*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, I. Caragiannis, M. Alexander, R. M. Badia, M. Cannataro, A. Costan, M. Danelutto, F. Desprez, B. Krammer, J. Sahuquillo, S. L. Scott, and J. Weidendorfer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, vol. 7640, pp. 175–184. [Online]. Available: http://link.springer.com/10.1007/978-3-642-36949-0_20

[55] J. Zeng, T. Matsunaga, and H. Mukai, "Using nvidia gpu for modelling the lagrangian particle dispersion in the atmosphere," 2010.

[56] A. Pinheiro, F. Desterro, M. Santos, C. Pereira, and R. Schirru, "GPU-Based Parallel Computation in Real-Time Modeling of Atmospheric Radionuclide Dispersion," in *Advances in Human Factors and*

System Interactions, I. L. Nunes, Ed. Cham: Springer International Publishing, 2017, vol. 497, pp. 323–333. [Online]. Available: http://link.springer.com/10.1007/978-3-319-41956-5_29

[57] M. C. D. Santos, C. M. N. A. Pereira, R. Schirru, and A. Pinheiro, "GPU-BASED PARALLEL COMPUTING IN REAL-TIME MODELING OF ATMOSPHERIC TRANSPORT AND DIFFUSION OF RADIOACTIVE MATERIAL," Jan. 2018. [Online]. Available: http://carpedien.ien.gov.br:8080/handle/ien/2145

[58] M. Govett, J. Rosinski, J. Middlecoff, T. Henderson, J. Lee, A. MacDonald, N. Wang, P. Madden, J. Schramm, and A. Duarte, "Parallelization and Performance of the NIM Weather Model on CPU, GPU, and MIC Processors," *Bulletin of the American Meteorological Society*, vol. 98, no. 10, pp. 2201–2213, Oct. 2017. [Online]. Available: https://journals.ametsoc.org/doi/10.1175/BAMS-D-15-00278.1

[59] J. Michalakes and M. Vachharajani, "GPU ACCELERATION OF NUMERICAL WEATHER PREDICTION," *Parallel Processing Letters*, vol. 18, no. 04, pp. 531–548, Dec. 2008, number: 04. [Online]. Available: https://www.worldscientific.com/doi/abs/10.1142/S0129626408003557

[60] C. El Amrani and I. M. Hedgecock, "Implementation of the WRF-Chem model in Grid Computing and GPU for regional air quality forecasting," in *2012 IEEE International Geoscience and Remote Sensing Symposium*. Munich, Germany: IEEE, Jul. 2012, pp. 2516–2519. [Online]. Available: http://ieeexplore.ieee.org/document/6350340/

[61] J. Mielikainen, B. Huang, H.-L. A. Huang, and M. D. Goldberg, "GPU Acceleration of the Updated Goddard Shortwave Radiation Scheme in the Weather Research and Forecasting (WRF) Model," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 5, no. 2, pp. 555–562, Apr. 2012, number: 2. [Online]. Available: http://ieeexplore.ieee.org/document/6155618/

[62] J. P. Silva, J. Hagopian, M. Burdiat, E. Dufrechou, M. Pedemonte, A. Gutiérrez, G. Cazes, and P. Ezzatti, "Another step to the full GPU implementation of the weather research and forecasting model," *The Journal of Supercomputing*, vol. 70, no. 2, pp. 746–755, Nov. 2014, number: 2. [Online]. Available: http://link.springer.com/10.1007/s11227-014-1193-y

[63] M. Huang, J. Mielikainen, B. Huang, H. Chen, H.-L. A. Huang, and M. D. Goldberg, "Development of efficient GPU parallelization of WRF Yonsei University planetary boundary layer scheme," *Geoscientific Model Development*, vol. 8, no. 9, pp. 2977–2990, Sep. 2015, number: 9. [Online]. Available: https://gmd.copernicus.org/articles/8/2977/2015/

[64] Y. Wang, Y. Zhao, J. Jiang, and H. Zhang, "A Novel GPU-Based Acceleration Algorithm for a Longwave Radiative Transfer Model," *Applied Sciences*, vol. 10, no. 2, p. 649, Jan. 2020. [Online]. Available: https://www.mdpi.com/2076-3417/10/2/649

[65] G. Ruetsch and M. Fatica, *CUDA Fortran for scientists and engineers: best practices for efficient CUDA Fortran programming*. Amsterdam : Boston: Morgan Kaufmann, an imprint of Elsevier, 2014.

[66] "NVIDIA HPC SDK Version 21.3 Documentation," May 2021. [Online]. Available: https://docs.nvidia.com/hpc-sdk/

[67] "Unified Memory for CUDA Beginners," Jun. 2017. [Online]. Available: https://developer.nvidia.com/blog/unified-memory-cuda-beginners/

[68] J. Kraus, "An Introduction to CUDA-Aware MPI," Mar. 2013. [Online]. Available: https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/

[69] A. Farguell, A. Corés, T. Margalef, J. Miro, and J. Mercader, "Data resolution effects on a coupled data driven system for forest fire propagation prediction," *Procedia Computer Science*, vol. 108, pp. 1562–1571, 2017. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1877050917305616

[70] W. Skamarock, J. Klemp, J. Dudhia, D. Gill, D. Barker, W. Wang, X.-Y. Huang, and M. Duda, "A Description of the Advanced Research WRF Version 3," UCAR/NCAR, Tech. Rep., Jun. 2008, artwork Size: 1002 KB Medium: application/pdf. [Online]. Available: http://opensky.ucar.edu/islandora/object/technotes:500

[71] "Conway's Game of Life - Rosetta Code," May 2021. [Online]. Available: https://rosettacode.org/wiki/Conway%27s_Game_of_Life#Fortran

[72] M. Gardner, "Mathematical games," *Scientific American*, vol. 223, no. 4, pp. 120–123, 1970. [Online]. Available: http://www.jstor.org/stable/24927642

[73] "Conway's Game of Life - LifeWiki." [Online]. Available: https://www.conwaylife.com/wiki/Conway%27s_Game_of_Life

[74] M. Kruse and T. Grosser, "DeLICM: scalar dependence removal at zero memory cost," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. Vienna Austria: ACM, Feb. 2018, pp. 241–253. [Online]. Available: https://dl.acm.org/doi/10.1145/3168815

APPENDIX

You must compile NETCDF with the same compiler that you will be compiling WRF-SFIRE with. Assume that the result is name netcdf-fortran and is in /usr/local/. Also assume that the Nvidia HPC SDK is installed in /opt/nvidia/hpc_sdk

Variable	Value
NETCDF_classic	1
WRFIO_NCD_LARGE_FILE_SUPPORT	1
NETCDF	/usr/local/netcdf-nvfortran/
LD_LIBRARY_PATH	/opt/nvidia/hpc_sdk/Linux_x86_64/21.1/compilers/lib

Table 7: Environment Variables

Command	Description
pgf90 life_v2.f90 -mp=gpu -Minfo=mp	Compiles a file called life_v2.f90 and looks for OpenMP directives into CUDA kernels to run on the GPU.
./compile -j 1 em_real & > compile.log &	Compiles the WRF code without multithreading and writes the output to compile.log
./compile -j 1 em_fire & > compile.log &	Compiles the SFIRE code without multithreading and writes the output to compile.log. The previous command must be run first.
mpirun -d -np 1 ./ideal.exe	After navigating to a test case, (test/em_fire/fireflux_small/ for example), execute the command to setup the test case. The parameters for the testcase are found in the namelist.input file in the same directory.
mpirun -d -np 6 ./wrf.exe	Run the executable with 6 MPI processes. This can only be run after running the previous command.
nsys profile mpirun -d -np 6 ./wrf.exe	Use Nvidia's Nsight Systems profiler to profile the code. This profiler gives an over all systems view of the execution.
ncu -o ncuprof –target-processes all -k module_fr_sfire_model_sfire_model_473_gpu mpirun -d -np 1 ./wrf.exe	Profile a specific kernel that was generated, in this case the module_fr_sfire_model_sfire_model_473_gpu kernel, using the Nvidia Nsight Computer profiler. This Profiler gives in depth information about the CUDA kernels.

Table 8: Useful Commands